MONOGRAFIES DE L'INSTITUT D'INVESTIGACIÓ
EN INTEL·LIGÈNCIA ARTIFICIAL
Number 1

Institut d'Investigació
en Intel·ligència Artificial

## Monografies de l'Institut d'Investigació en Intel·ligència Artificial

| | |
|---|---|
| Num. 1 | J. Puyol, *MILORD II: A Language for Knowledge–Based Systems* |
| Num. 2 | J. Levy, *The Calculus of Refinements, a Formal Specification Model Based on Inclusions* |
| Num. 3 | Ll. Vila, *On Temporal Representation and Reasoning in Knowledge–Based Systems* |
| Num. 4 | M. Domingo, *An Expert System Architecture for Identification in Biology* |

# MILORD II: A Language for Knowledge–Based Systems

Josep Puyol i Gruart

Foreword by Jaume Agustí

Institut d'Investigació en Intel·ligència Artificial
Bellaterra, Catalonia, Spain.

Series Editor
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Foreword by
Jaume Agustí
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
Josep Puyol i Gruart
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Institut d'Investigació
en Intel·ligència Artificial

ISBN: 84–00–07499–8
ISSN: 1135–4100
Dep. Legal: B–34740–95
© 1995 by Josep Puyol i Gruart

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.
**Ordering Information:** Text orders should be addressed to the Library of the IIIA, Institut d'Investigació en Intel·ligència Artificial, Campus de la Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain.

Printed by Cardellach Còpies, S.L. CBS, S.A.
Sant Pere, 40.
08221 Terrassa, Spain.

*Als meus pares.*
*A Carme i Pau.*

# Contents

**Foreword** **xi**

**Preface** **xiii**

**Abstract** **xv**

**1 Introduction** **1**
1.1 Motivation . . . . . . . . . . 1
1.1.1 Real World Expert Systems . . . . . . . . . . 1
1.2 From *Milord* to **Milord II** . . . . . . . . . . 3
1.2.1 *Milord* Characteristics . . . . . . . . . . 3
1.2.2 Differences and Improvements . . . . . . . . . . 4
1.3 Related Work . . . . . . . . . . 6
1.3.1 Purpose . . . . . . . . . . 7
1.3.2 Modularity . . . . . . . . . . 7
1.3.3 Approximate Reasoning . . . . . . . . . . 8
1.3.4 Inference Engines . . . . . . . . . . 9
1.3.5 Control . . . . . . . . . . 10
1.4 Main Contributions . . . . . . . . . . 10
1.5 Scheme of the Thesis . . . . . . . . . . 13

**2 Modularity** **15**
2.1 Introduction . . . . . . . . . . 16
2.1.1 Previous Work . . . . . . . . . . 17
2.1.2 Modular System . . . . . . . . . . 17
2.2 Primitive Components . . . . . . . . . . 19
2.2.1 Interfaces of modules . . . . . . . . . . 22
2.2.2 Modular hierarchy . . . . . . . . . . 24
2.2.3 Semantics of modules . . . . . . . . . . 25
2.3 Generic modules . . . . . . . . . . 28
2.4 Refinement, Expansion and Contraction . . . . . . . . . . 32
2.4.1 Refinement . . . . . . . . . . 34
2.4.2 Expansion and Contraction . . . . . . . . . . 40
2.5 Special declarations . . . . . . . . . . 41

2.5.1 Inherit and Open . . . . . 41
2.5.2 Sharing . . . . . 43
2.5.3 Dynamic Modules . . . . . 43
2.6 Conclusions . . . . . 44

**3 Approximate Reasoning 45**
3.1 Algebra of truth–values . . . . . 47
3.1.1 Modus Ponens Operator . . . . . 50
3.2 Uncertainty and Imprecision . . . . . 51
3.2.1 Intervals of Truth–values . . . . . 54
3.2.2 Working with intervals . . . . . 56
3.2.3 Fuzzy Sets . . . . . 57
3.3 Local Logics . . . . . 60
3.3.1 Mappings between different local logics . . . . . 61
3.3.2 Example . . . . . 63
3.4 Logic Declaration . . . . . 65
3.4.1 Truth values . . . . . 65
3.4.2 Connectives . . . . . 66
3.4.3 Renaming . . . . . 67
3.5 Conclusions . . . . . 68

**4 Deduction by Specialization 71**
4.1 Enriched Behavior . . . . . 71
4.1.1 Communication . . . . . 73
4.1.2 Solutions . . . . . 74
4.1.3 Validation . . . . . 79
4.1.4 Summary . . . . . 80
4.2 Specialization Calculus . . . . . 81
4.2.1 Syntax . . . . . 81
4.2.2 Semantics . . . . . 82
4.2.3 Specialization Calculus . . . . . 86
4.2.4 Soundness and Completeness . . . . . 87
4.3 Implementation . . . . . 87
4.3.1 Inference Engine Design . . . . . 88
4.3.2 Internal Representation of Deductive Knowledge . . . . . 89
4.3.3 Specialization . . . . . 90
4.4 The Deductive Knowledge Language . . . . . 94
4.4.1 Facts . . . . . 95
4.4.2 Rules . . . . . 100
4.4.3 Predicates on Facts . . . . . 101
4.5 Conclusions . . . . . 105

**5 Control** **107**
5.1 Implicit Control . . . 109
5.1.1 Subsumption . . . 109
5.1.2 Unnecessary Rules . . . 115
5.2 Threshold . . . 116
5.3 Evaluation Strategy . . . 117
5.3.1 Lazy . . . 117
5.3.2 Eager . . . 119
5.4 Reification and Reflection Mechanisms . . . 119
5.4.1 Static Reification . . . 121
5.4.2 Dynamic Reification . . . 122
5.4.3 Deductive Control . . . 124
5.4.4 Structural Control . . . 124
5.5 Conclusions . . . 126

**6 Applications** **127**
6.1 Introduction . . . 127
6.2 Terap-IA . . . 128
6.2.1 Motivation and Goals . . . 128
6.2.2 Architecture . . . 128
6.2.3 Implementation . . . 131
6.3 *Spong–IA* . . . 136
6.4 *Ens–AI* . . . 136
6.5 Fuzzy Control Example . . . 139
6.5.1 Simulation Process . . . 140
6.5.2 Controller . . . 141
6.5.3 Results . . . 145
6.6 Propagation Rules for Polytrees . . . 146
6.6.1 Introduction . . . 146
6.6.2 Implementation over **Milord II** . . . 150
6.7 Future Applications . . . 155
6.8 Conclusions . . . 155

**7 Conclusions** **157**
7.1 Future Work . . . 158

**A Syntax of Milord II** **161**
A.1 Notation . . . 161
A.2 Modular System . . . 163
A.3 Deductive Knowledge . . . 164
A.3.1 Dictionary . . . 164
A.3.2 Rules . . . 164
A.4 Inference System . . . 166
A.5 Control Knowledge . . . 167
A.5.1 Evaluation Type . . . 167
A.5.2 Truth Threshold . . . 167

A.5.3 Deductive Control . . . . . . . . . . . . . . . . . . . . . . . 167
A.5.4 Structural Control . . . . . . . . . . . . . . . . . . . . . . . 167

**B Proofs** **169**
B.1 Proposition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 169
B.2 Soundness Theorem . . . . . . . . . . . . . . . . . . . . . . . . . 171
B.3 Restricted Completeness . . . . . . . . . . . . . . . . . . . . . . 171
B.3.1 Literal Completeness . . . . . . . . . . . . . . . . . . . . . . 171
B.3.2 Restricted Literal Completeness Theorem . . . . . . . . . . 174

**C Code Examples** **177**
C.1 *Terap–IA* Example . . . . . . . . . . . . . . . . . . . . . . . . . 177
C.2 Fuzzy Control Example . . . . . . . . . . . . . . . . . . . . . . . 195
C.2.1 Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . 196
C.2.2 Simulator . . . . . . . . . . . . . . . . . . . . . . . . . . . . 202
C.2.3 Whole Process . . . . . . . . . . . . . . . . . . . . . . . . . 202
C.3 Polytrees Example . . . . . . . . . . . . . . . . . . . . . . . . . . 204

**List of Figures** **209**

**List of Tables** **211**

**References** **213**

**Index** **221**

# Foreword

The Expert System Shell presented in this book, **Milord II**, is the result of a long and intensive research effort made during eight years within the *IIIA* in developing several *real* life Expert Systems. **Milord II** has been designed and implemented not only having some possible usages in mind, but during and through the applications development. To obtain the computational tools that do some tasks previously done by professionals. So **Milord II** although it generalizes from the particular domains who guided its design – mainly medicine – it should be best understood and be most useful for the generic tasks it was thought for, classification problem solving. This specialization, the narrow and deep adaptation to a kind of problem, I think is a mark of good engineering. Practical engineering including software engineering should be domain driven.

The author of the book fits perfectly well into this dynamic pattern of practical, day to day engineering. The practical solutions he gave to the problems were frequently ahead of the theoretical reflection and the foundational effort. Because the book is the Ph.D. Thesis of the author this practical trend is made less evident than it could be. Only the last chapter on applications is completely devoted to show it. The four central chapters of the book show the theoretical and technical foundations of **Milord II**. Firmly grounded on them, **Milord II** is a powerful tool for classification problem solving with uncertain and incomplete information, allowing modular and incremental development and reuse of solutions.

Bellaterra, February 1996

Jaume Agustí
Head of the
Formal Methods Department
of the *IIIA*, CSIC

# Preface

One of the main topics at the *IIIA* has been the study and development of Knowledge–Based Systems, going from the theoretical aspects to the practical development of languages for Expert Systems and real world applications.

Since 1985 our group has been working on the development of a shell for Expert System named *Milord*. The first version of *Milord* was finished in 1989. *Milord* introduced great important advances on uncertainty management languages and multi–level architectures for Expert Systems. The main applications developed using *Milord* were in the medical domain. The most important was *Pneumon–IA*, an Expert System for the diagnosis of pneumoniae.

The acceptance of *Milord* shell and its applications prompted us to think in a second version much renewed of the shell we named **Milord II**. Since 1989 we have been developing **Milord II**, which is the topic of the present work.

As it can be seen from the structure of this book the new contributions have been the modularization, the uncertainty management, the deduction by specialization and the reflective control architecture of **Milord II**. Like in *Milord*, this research has been driven by the applications, then a set of new real world applications and examples have been developed using **Milord II**. In this book we present some Expert Systems on different domains: *Spong–IA*, for identification of marine sponges; *Terap–IA*, for treatment of pneumoniae (the natural extension of *Pneumon–IA*); *Ens–AI*, for psicopedagogical diagnosis; and some small examples.

The research on **Milord II** is not considered to be finished in the sense that new applications and ideas still contribute to the continuous enrichement of the language. Now we are working on a new version of **Milord II** (MilordAgents ?) based on Multi–Agent Systems. The main idea is to study the cooperation of cognitive agents based on **Milord II** modules.

**Milord II** has been developed in Common Lisp (the interpreter) and in C (the compiler). This software is available for research and educational purposes. A fresh version for Macintosh[1] machines can be obtained by *anonymous ftp* at ftp.iiia.csic.es in the directory /pub/Milord/mac, or in the WWW at http://www.iiia.csic.es/˜puyol. Now we are working on versions for PC and Unix machines.

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

# Acknowledgments

This work has been influenced by many people. I specially thank Jaume Agustí, who introduced me to research activities and who has provided guidance and support in the development of this work.

Carlos Sierra has greatly influenced this work because this thesis is the natural extension of his previous work, *Milord*. He has provided me extensive support. We had large discussions on **Milord II** and I have benefited of his advice and assistance during **Milord II** design and application development.

Lluís Godo has collaborated in all the questions dealing with uncertainty management.

I am in debt with the experts that have applied **Milord II** to real world problems. Pilar Barrufet, Marta Domingo, Clara Barroso and Lluís Murgui have been patient and constant users of my system.

The Milord II Compiler has been developed by Josep Lluís Arcos. He has suffered all the continuous changes we were introducing in the language.

Finally I would thank all the *IIIA* colleagues and friends for their collaboration and support, specially Francesc Esteva, *IIIA* Director, and Ramón López de Mántaras, head of our group.

All this work has been developed first in the Artificial Intelligence Group at the Center for Advanced Studies of Blanes (CEAB), which in October 1995 moved to the newly created Artificial Intelligence Research Institute (*IIIA*). CEAB and *IIIA* are research institutes from the Spanish Scientific Research Council (CSIC). This work has been financed mainly by CICYT Spanish projects, SPES project n. 880j382 and TESEU project TIC91–0430. My thanks to these Institutions for providing the necessary means for the development of this work.

Bellaterra, February 1996

Josep Puyol i Gruart
E–mail: puyol@iiia.csic.es
http://www.iiia.csic.es/~puyol

# Abstract

**Milord II** is an architecture and a language for the development of knowledge–based systems. In particular we are interested in real world Expert Systems, that is, those that are useful in a real environment and that have real purposes. To do that we propose a language based on modules as a method for programming in the large. Modules, generic modules and a set of operations on them are the basis of this language. A program in **Milord II** is then a hierarchical structure of modules. A module is an encapsulated unit with a well defined interface to other modules. Each module is composed of deductive knowledge (weighted facts and rules), local logic (a truth values algebra declaration) and a local control component (Horn–like metarules).

Each module contains its own local logic to deal with approximate reasoning. An algebra of truth–values is defined to perform the deductions in a weighted rule–based language (deductive knowledge). A mechanism is provided to find valid translations of the terms communicated between modules with different logics.

The deduction mechanism of **Milord II** is based on the concept of Specialization. This leads to a new inference engine which improves the deduction compared with the engines based on Modus ponens. These improvements facilitates the communication with the user, the validation and the understanding of Experts Systems.

Finally we discuss a set of applications and examples developed using **Milord II**.

# Chapter 1

# Introduction

To introduce the content of this thesis it is necessary first to talk about what are the problems we want to handle, and what is the kind of solutions we propose. Here it is very important to fix the type of problems, its environment, what kind of solutions we are interested in, where, and who is the user of those solutions. In this Chapter the motivation, the history and a summed up description of **Milord II** modular language and its environment are presented.

## 1.1 Motivation

We say that somebody is an *expert* when he is skilled in some matter by practice. Examples of *human experts* are physicians, biologists, mechanics, engineers, and so on. They are able to solve problems by using knowledge obtained by practice, despite they also have somewhat deep knowledge about their knowledge domain. Expert Systems (ESs) have proved to be useful tools to automate this kind of problem solving.

The goal of this thesis has been the design and implementation of a modular language named **Milord II**, that offers a powerful, simple and friendly environment to develop ESs. We should notice that the starting point of this thesis was the previous language *Milord*. This language and their applications allowed us to experience new problems, which guided us to the design of a new language based on the previous one.

First we should fix what kind of Expert Systems **Milord II** is intended for, and who are the expected programmers and users of **Milord II** and its applications.

### 1.1.1 Real World Expert Systems

One of the main characteristics of **Milord II** is that it addresses the development of real world ESs, that is, the problems we want to solve are not *toy* examples,

and both the programmers and the users are professionals interested in obtaining good results from the system.

Below we present the main characteristics of our work environment, that lead us to the actual design and implementation of **Milord II**.

- We think that the *programmers* of ES applications with **Milord II** are experts in some knowledge domain. Normally they are not knowledge engineers or artificial intelligence specialists. In general, they deal with application domains where *expertise* is required, for instance medical or biological domains. That means that they need simple tools and simple languages to develop their applications[1]. **Milord II** has been designed to be an easy to use system.

- Experts are qualified and busy professionals, then we should offer friendly tools to them. Simplifications of problems would lead us to good examples, but the interest of experts in them will be poor. This implies the need to handle non simplified real problems to motivate the use of the system by experts. In this case programming an ES becomes a useful work being able to structure, understand and diffuse the own knowledge the expert has. Experts have used **Milord II** to develop ES applications as biological classification, medical diagnosis, etc.

- Notice that our applications are *highly interactive* and that this interaction is done with humans. We consider *users* of **Milord II** people who works with the ESs generated by programmers. *Users* have different *levels of expertise*. They can be experts in the domain knowledge of the system, or non expert users. It both cases they need a good *communication* with the system and a high level of *confidence* on it. Specialization is a key concept introduced in **Milord II** to produce a good communication with the users.

- Some features of the *real applications* are the big size, and the incomplete, imprecise and uncertain knowledge they have. To match these characteristics we needs an *expressive language* that provides structuration tools, incremental design, reutilization of components and approximate reasoning. These characteristics are very important in the design of **Milord II**. Modularity and uncertainty treatment are the key points of the system together with specialization.

- Most of the problems we want to handle are *classification tasks*. We consider classification the task of finding solutions in a known and finite search space. *Milord* has been applied mostly to this kind of applications, and **Milord II** follows the same way. Examples of classification applications are medical diagnosis, biological classification, and so on.

[1] Considering the experts as programmers do not mean that applications are totally developed by experts. In our case experts have continuous advising from us, we take then the role of knowledge engineer. It is easy to see that if experts have a good knowledge of the tools they are using, the communication between the expert and the knowledge engineer is easier.

- Finally we think that real applications should work in *real environments*. The lack of informatic resources in the environments where the applications could be tested and used is a common problem. This contributes to isolate the system. We should accommodate and we did it the resource requirements and the efficiency of **Milord II** shell to the most normal machines usually present in the environment of the application.

We have explained the main features and problems that experts and users[2] of Real ESs must face. In this thesis we will propose solutions for that. At this point we should talk about the previous experience with *Milord* and the improvements that **Milord II** introduces to handle the kind of ESs described above.

## 1.2 From *Milord* to Milord II

Notice that the development of **Milord II** was possible thanks to our previous experience acquired in ES design. The acceptance of *Milord* Shell (Godo et al., 1988; Sierra, 1989) and the applications developed allowed us to think in a second version much renewed of the Shell we named **Milord II**. This experience and all the considerations above lead us to the design and implementation of **Milord II** Shell, keeping in mind that the programs should be useful in the real environment of the application domain.

The transition from *Milord* to **Milord II** was a natural one directed by the applications. The main applications developed with *Milord* were in the medical environment, as Pneumon–IA (diagnosis of pneumonia) (Verdaguer, 1989) and RENOIR (rheumatic disease diagnosis) (Belmonte, 1991). Furthermore *Milord* was the starting point of new works on validation (Meseguer, 1992) and case–based reasoning (López, 1993).

The new applications we are developing with **Milord II** are: *Pneumon–IA II* (a modular version of *Pneumon–IA*, a system for pneumonia diagnosis), *Terap–IA* (treatment of pneumonia), *Spong–IA* (sponge classification) and *Ens–AI* (psycho–pedagogical diagnosis).

To see the improvements and differences of **Milord II** with respect to the previous version we summarize *Milord* characteristics in the following. The needs detected during the development of the applications have guided the improvements of the system.

### 1.2.1 *Milord* Characteristics

It is not our purpose here to describe exhaustively the *Milord* system. Here we want to point out those elements that had a close relation to the new design. We can summarize the main characteristics of *Milord* in the following points:

[2]Notice that in the following, when there is no ambiguity, we will use *experts* meaning the programmers of **Milord II** applications. When it is necessary we will distinguish between *users* and *expert users*.

**Multi–level Architecture:** One of the main purpose of *Milord* was the clear separation among different sorts of knowledge, that is, associative, structural, hypothetic and heuristic. That architecture allowed the experts to give a good structuration of his knowledge providing a clear separation between domain and control knowledge.

**Uncertainty Treatment:** A good effort was done to approach and represent the type of uncertainty the experts normally use to express their knowledge. *Milord* used fuzzy logic with linguistic labels to represent uncertainty. Experts were able to define their own logic in the applications.

**Modularity:** A first proposal (design not implementation) of modularity was done in *Milord* in order to support programming in the large. This proposal was based in the modularization of the domain knowledge only as a static structuration tool. Before the interpretation of a program, it had to be compiled to a *flat* structure, that is, to an equivalent and non modular program. The multilevel architecture of control was outside this modular structuration.

**Communication:** The behavior of *Milord* was the *standard* of many Expert Systems. The user gives a goal to the ES, and the ES asks to the user for the values of the facts which are relevant to obtain a solution. Finally the ES answers the value of the goal, or *unknown* if it was not able to obtain a solution.

**Implementation:** The version of *flat*[3] *Milord* was implemented and tested with the applications mentioned above. Because of the great amount of resources that symbolic computation consumes (*Milord* was programmed in Common Lisp), the current technological state forced the use of *mini* computers to implement and run *Milord*.

Following the above points we can give a brief analysis of the main differences and improvements of **Milord II** with respect to *Milord*.

### 1.2.2 Differences and Improvements

*Milord* and **Milord II** have many common points, going from the characteristics of the language to the applications programmed with them. It is very difficult to analyze in depth these points without making an exhaustive explanation of both systems. For that reason here we only describe the main conceptual and architectural differences between *Milord* and **Milord II** leaving the details of the common points and the differences to be explained along this thesis. The points we will take into account are: modularity, approximate reasoning, behavior and communication. A summary of these differences between *Milord* and **Milord II** are given in the Table 1.1.

[3]Modular *Milord* was not implemented.

| | *Milord* | **Milord II** |
|---|---|---|
| structuration | multi–level architecture | modularity + local control |
| control | propositional meta–rules | first–order meta–rules |
| uncertainty | linguistic labels | intervals of linguistic labels |
| logic | default logic | local logic to modules |
| inference engine | backward, forward | specialization |
| communication | standard behavior | enriched communication |

Table 1.1: Main differences between *Milord* and **Milord II**.

### Modularity and Control

The first point we studied in the design of **Milord II** was the modularity. The final proposal of modularization was more radical than that of *Milord* in the sense of encapsulating all the components of an ES into modules. A module contains both the object level and the control level. There are no global components in **Milord II**.

The first consequence of that decision is that the control is local instead of global. We consider that control is a component of the problem solving task tied to the domain knowledge. Modularity allows us to decompose a problem into simple subproblems. Following this kind of structuration of problems, it is easy to see that, when we decompose a problem, we know how to control its subproblems. Global control would confuse this kind of structuration.

The second consequence is that we substitute the multilevel architecture of control of *Milord* with a structure composed by a hierarchy of modules with local control. It is well known that we can think in multiple levels of control. Finally the question "Who controls the control?" is frequent. Despite the multilevel control of *Milord* showed to be useful in the applications, it is limited. The modules with local control of **Milord II** are more flexible, they allow us to build other multilevel architectures with the numbers of levels required. The critical point is that there is no a preprogrammed architecture of control as in *Milord* and then the programming task could be more difficult.

From the implementation point of view another difference between **Milord II** and *Milord* is that, before run time, *Milord* compiles the modular object level of the program to a *flat* one. **Milord II** preserves the modular structure, the modules are objects with a permanent entity.

Another characteristic of **Milord II** is that it provides dynamic modules, that is, the modular structure can be created at run time.

### Approximate Reasoning

After the definition of the modular language, the second topic that was treated in the design of **Milord II** was the approximate reasoning.

Programmers of *Milord* were able to define the logic that would be used in

their applications. Taking into account the modular structure of **Milord II** we studied the possibility of defining different logics into the different modules. This means we can use different languages of representation in the different modules (Harper et al., 1989). Then, in **Milord II** the definition of logics is local to the modules and it provides mechanisms of communication between the different local logics of the modules.

Another improvement of **Milord II** over the uncertainty treatment of *Milord* is the introduction of imprecision. This is made by means of the extension of the uncertainty calculus of linguistics labels to the intervals of linguistics labels, and the use of fuzzy sets.

### Behavior and Communication

One of the main goals of this thesis is to enrich the behavior of ESs. A standard ES receives queries from the user, asks questions to the user, and finally answers the queries of the user. We are interested in improving the way the system ask the user, and in enriching the sort of answers the system gives to the user. Remember that normally our applications are interactive and the users are human. Then the following points are very important:

- It should be clear to the user that the sequence of questions he is answering actually drives the system to the solution he is interested in, and that the information he gives to the system is properly used to find this solution.
- The solutions found by the system to the queries of the user should be informative enough. For instance, the answer *unknown* is not informative at all, it only says that the system was not able to find a solution.

To solve these problems **Milord II** introduces an inference engine based on specialization of KBs. This kind of inference engine allows us to make independent the search and the deductive processes instead of the classical interleaved search and deductive processes of the standard inference engines as backward and forward ones. This allows us to implement different control strategies in order to improve the quality of the process of obtaining information from the user. Furthermore the inference engine of **Milord II** is able to obtain conditional answers and deal with *unknown* answers from the user.

Finally notice that **Milord II** Shell runs over *personal* computers to facilitate its use by experts in the environment of the applications. This is not a merit of the implementation but of the new advances introduced in the current personal computers.

## 1.3 Related Work

After the first experiences with ESs at seventies (for instance, MYCIN (Shortliffe, 1976)) knowledge–based systems, and artificial intelligence in general, have experimented a continuous evolution of the ideas, styles and techniques.

Artificial Intelligence have been more and more specialized and a great number of new disciplines have come out. Knowledge representation, problem solving methods, approximate reasoning, methodologies for knowledge engineering, formal methods, learning, and so on, are examples of topics that have generated a lot of work.

We will give a brief explanation of some aspects we consider they are the most relevant to relate with **Milord II**. The languages appeared in this Section are from the classical ones to other actual languages that we consider are interesting to be related with **Milord II** now, and in the future. You can find a very interesting description of some of these systems (including **Milord II**) through a common example in (Treur and Wetter, 1993). In the same book there is a comparison study of these languages (vanHarmelen et al., 1993).

### 1.3.1 Purpose

The first criteria for the comparison of languages is the purpose the languages pursue. We can distinguish between the languages that are designed to build executable systems for some concrete applications from those that find formal specifications of general tasks, problem solving methods, domain models and so on.

The purpose of the language determines the sort of development task used to design and implement a concrete language for knowledge engineering. The languages directed to the applications are designed following a bottom–up methodology. There is a feedback cycle between the language designers and the experts with their applications. Then, this kind of languages are incrementally designed and implemented on demands of the applications and the experts. The second type of languages are devoted to the modelization of more general problems and they follow a top–down methodology of development.

The resultant languages designed with these approaches differ on their expresivity power. Those languages designed with the first type methodology are more expressive and easy to use for the specific kind of problems they has been designed. The others can cope a wider set of problems, but the expresivity power becomes poor.

As explained in the introduction, we are interested in the implementation of real ESs, then the development of **Milord II** was directed by the applications it was involved in. We have designed a language the more adapted to the kind of problems it is applied. AIDE (Gréboval and Kassel, 1992) language has a similar approach to develop real ESs and it is oriented to give good explanations. **Milord II** is able to build real size applications.

### 1.3.2 Modularity

All the language designers agree with the need of providing programming constructs for the modularization of the programs. Some languages encourage more than others this technique and the architectures of the modular systems are different.

In some cases the current methodologies for knowledge engineering, as components of expertise (Steels, 1990), KADS (Wielinga et al., 1992) or generic tasks (Chandrasekaran, 1987), determine the kind of modularization used in the languages.

Several languages are related with KADS methodology such as $(ML)^2$ (van-Harmelen and Balder, 1992), AIDE (Gréboval and Kassel, 1992), KARL (Fensen et al., 1991), and $K_{BS}SF$ (Veld et al., 1993). These systems implement the global layering of KADS methodology, that is, domain, inference, task and strategy layers. The modularization is limited to be used into each layer, i.e. a modular structure in the task layer or in the strategic layer.

The language COMMET (Jonckers et al., 1992) uses the components of expertise methodology based on tasks, models and methods.

Other languages as DESIRE (Langevelde et al., 1993) and the language MC (Giunchiglia et al., 1993) do not have this limitation and the specifications are a set of interconnected reasoning modules, where each module is treated as an independent unit. These languages are used to define different kind of modules, such as domain and control modules.

The approach of **Milord II** is different in the sense that each module contains the domain knowledge (object level) and the local control knowledge (meta level) of the module. The interaction between modules is limited to object–object only, thus forcing purely local application of the meta–knowledge.

No one of the previous modularization techniques is based on the idea of a module as a specialist like in **Milord II**.

Mostly of the languages (including **Milord II**) encourage the user to encapsulate the local knowledge into modules. Other systems as AIDE has programming constructs but they do not force the user to exploit them. The modularization techniques of **Milord II** are based on theories used in the language ML (Harper et al., 1986).

### 1.3.3 Approximate Reasoning

Usually the information contained in the KBs is imperfect. Experts manage uncertain, imprecise, and incomplete information. Approximate reasoning is then an important topic in the development of ESs.

There are three main approaches to deal with uncertainty, that is, the probabilistic, the evidential, and the possibilistic approach. Let us to give a brief vision of these approaches in order to situate that of **Milord II**.

There are several models based on probability. We can consider Bayesian Networks (Pearl, 1986), Nilsson's Probabilistic Logic (Nilsson, 1986), Subjective Bayesian Networks of PROSPECTOR (Duda et al., 1976), and Certainty Factors of MYCIN (Shortliffe and Buchanan, 1975).

The lack of expressiveness is the main problem of the models based on probability. They can not express vague predicates (for instance, tall). We must specify probabilities but in practice they represent subjective appreciations that are not based on statistical analysis. In this case it is very difficult that experts are able to represent these probabilities with enough precision by means

of real numbers. Finally notice that the models based on bayesian networks are computationally expensive.

The evidential theory of Dempster–Shafer (Dempster, 1967; Shafer, 1976) has its main problem in the computational complexity, despite it is very useful to manage uncertainty.

Both *Milord* and **Milord II** are based on possibilistic approaches. Zadeh introduced fuzzy logic to manage uncertainty and vagueness (Zadeh, 1975). We think that this approach provides an understandable and computationally efficient method to deal with imperfect information. *Milord* uses a linguistic approximation based on linguistic terms as fuzzy intervals (Godo et al., 1988; Sierra, 1989). The expert declares the linguistic terms as fuzzy intervals by defining a trapezoidal characteristic function. From these declarations *Milord* computes the truth tables of the conjunction, disjunction and implication operations. In the *Milord* approach still remains the problem of the numerical representation of the linguistic terms.

**Milord II** uses multi–valued propositional logic. The expert can choose a set of linguistic terms and define a set of logical operations directly on the set of linguistic values.

**Milord II** uses this kind of logic because multi–valued propositional sentences are easy to understand and to use for the kind of applications we are normally involved. The lack of first order constructs is compensated by multi–valuedness of the logic (for instance, DESIRE use three–valued first order logic).

It is very important to notice that we extend the multi–valued logic to intervals of linguistic terms and that the modules of **Milord II** contain its own local logic (Agustí et al., 1991; Agustí et al., 1992) . **Milord II** provides the constructs of the language and a set of utilities to define different local logics adapted to the different problems represented by different modules. Furthermore we provide the method to find valid mappings between these logics in order to communicate different modules with different local logics without loss of consistency.

### 1.3.4 Inference Engines

A lot of systems based on first order logic use the Prolog technology. In other cases the classical inference engines like that backward and forward ones are used. For instance, Teiresias (Davis, 1982) has simple control strategies based on forward or backward engines.

As cited above the inference engine used in the modules of **Milord II** is based on the specialization of KBs (Puyol, 1992a; Puyol et al., 1992b). Specialization is based on the notion of *partial evaluation* expressed in the well known Kleene's Theorem (Kleene, 1952). Briefly, if we have a function of $n$ arguments and we know the value of an argument we can specialize this function obtaining a new one with the same arguments that before but the known one. We can consider a KB as a function with arguments the set of facts needed to reach the goal of the KB. We specialize the KB with a known fact obtaining a new KB specialized by the new domain that contains the known fact.

**Milord II** is based on logic, then we use the term *partial deduction* instead of partial evaluation following the suggestion of Komorowski (Komorowski, 1981; Komorowski, 1990). Partial deduction algorithms have been intensively used in logic programming (Venken, 1984; Gallagher, 1986; Komorowski, 1981; Takeuchi and Furukawa, 1986; Lloyd and Shepherson, 1991) mainly for efficiency purposes. Our approach is different for instance from the logic programming one used in (Lloyd and Shepherson, 1991). There, partial evaluation was goal driven, whereas here partial evaluation is data driven.

**Milord II** inference engine is also related to other work on *conditioned answers* (Demolombe, 1990; Vasey, 1986; Sakama and Itoh, 1986) and on the treatment of *unknown information* (Wolstenholme, 1987). Specialization used in **Milord II** allows us to obtain conditioned answers after the specialization of a KB with the known information. Our system is able to answer a useful result even in the case of partially unknown information.

The main difference of **Milord II** specialization with respect to other uses of partial deduction, is that it is based on a multi–valued propositional language and it is oriented to the improvement of the communication of ESs.

### 1.3.5 Control

The first type of production rule languages like OPS5 (Forgy, 1981) used to define a single level of production. **Milord II** as DESIRE have declarative control through a reflection mechanism. Other systems use procedural, functional, or *guided by the user* control.

**Milord II** uses Horn–like rules to define the control knowledge of a module. Remember that the interaction between modules is limited to object–object interaction, then the meta–knowledge is local to each module (DESIRE and MC have metamodules).

**Milord II** does not use global control and all the components of control are local to the modules. This allows us to clarify the problem structuration giving a easy idea of the way control is implemented in an application.

## 1.4 Main Contributions

The main contribution of this thesis is the integration and implementation of a set of techniques, like specialization in multi–valued logics, and theoretical results, some of them introduced here, into a language for the development of ESs. The language has been implemented and a set of real size applications has been developed.

The structure of this thesis is aimed at a deep, exhaustive and understandable description of the **Milord II** system. Then we summarize the contributions of the thesis following the same scheme of the thesis Chapter by Chapter.

**Chapter 2.** *A modular language which allows a top–down and incremental methodology of ESs programming.*

Classical software engineering approves top–down design as a good programming methodology. The modular language of **Milord II** allows experts to develop programs with a disciplinated methodology based on the decomposition of problems into simpler subproblems.

Each module of **Milord II** contains all the components that usually define a whole ES, that is, the domain knowledge, the control knowledge, the logic and so on. The idea of a module as a local expert or specialist distinguishes our modular system from others. Then we can consider an ES as composed of a set of ESs modules, one for each subproblem to solve. Modules are organized into a hierarchical structure that provide the form of integrating the solutions of the subproblems to build the solution of the whole problem.

Incremental programming is another interesting feature of top–down programming. It consists in defining problems at different level of detail, initially by means of a partial description which is successively refined obtaining more concrete definitions of the problem. We stop when the level of detail is the required one.

The modules have a well defined interface, and the language also provides generic modules and a set of operations devoted to the incremental programming of modules.

The contributions of this Chapter to the development methodology of knowledge–based systems are the following:

- A methodology of programming based on problem decomposition.
- An homogeneous language based on modules, generic modules and operations between them. The primitive component of **Milord II** are the modules. Relational, functional, logic, and control knowledge, are encapsulated into modules. All the components are local to modules.
- A methodology of incremental programming for knowledge–based systems.

**Chapter 3.** *Managing imperfect information. Local logics.*

The knowledge that experts manage is imperfect, that is, incomplete, imprecise or uncertain. **Milord II** deals with this kind of information by means of an object level language of order $0^+$ based on many–valued logics. The use of linguistic terms as truth–values makes the language closer to the experts.

**Milord II** allows the experts to define the local logic that will be used in each module, this allows us to use in each module the logic more adequate to the problem that the module represents. The expert can also define the mapping between the terms of the different local logics of modules. A mechanism to find valid mappings between these terms is provided. The terms of a module are in this way put into correspondence with the different terms of another module.

The contributions of this Chapter are of different types:

- An empirical contribution to the use of multi–valued logics based on intervals of truth–values to deal with uncertainty and imprecision.
- An empirical and development methodology contribution by the introduction of fuzzy sets into the data types of **Milord II**.
- A contribution to the theory of local logics mappings and a practical formulation of the algorithms to find morfisms between different logics.
- A development methodology contribution by the introduction in the language of the local logic declarations and the translations of terms of different modules with different logics.

**Chapter 4.** *A new behavior of ESs based on Specialization of Knowledge Bases.*

We consider a module as an entity capable of solving a concrete problem in a well defined domain. Then we say that a module is a specialist. When we introduce a new piece of information into a module we are specializing the module for a new and more restricted domain (the previous one plus the new information). The inference engine for the object level language of **Milord II** is based on this concept of specialization. As we will see this gives us an enriched behavior that allows conditioned answers.

The contributions of this Chapter are of different types:

- An empirical contribution to the interpretation of deductive processes as the specialization of KBs, and its applications to the improvement of the whole behavior of an ES.
- A theoretical contribution to the development of the *Specialization Calculus* or deduction by specialization with uncertainty.
- An empirical contribution to the separation of the control and logic semantics of the deduction by the separation of the search and deductive processes in the inference engine.
- An empirical contribution to the design of the deductive process.
- A development methodology contribution by the definition of a language to declare the deductive knowledge of modules.

**Chapter 5.** *Control adapted to the modular structure.*

**Milord II** has no global components, all the components are encapsulated into modules. That is the same for control. When we decompose a problem into subproblems we rely on strategies to focus the problem solving behavior to the more adequate subproblem in each moment. Local control allows us to define a set of control parameters and a set of metarules. These metarules control the execution of the module that contains them and also are responsible of the hierarchical structure of the submodules.

We can summarize the contributions of this Chapter:

- An empirical contribution to the control. Implicit control takes advantages of the specialization allowing to save questioning, computation and using the more specific knowledge.
- A development methodology contribution by introducing the explicit local control into modules. It is composed of static declarations (threshold, evaluation) and dynamic ones based on Horn–like metarules.

**Chapter 6.** *A set of applications developed using* **Milord II**.

The applications developed using **Milord II** and the set of new problems they have raised ensures that **Milord II** can be used to model complex problems that belong to the category of real ESs.

- Contributions to the development methodology by advising and giving support to experts during the development of real applications and some examples.

## 1.5 Scheme of the Thesis

This thesis is composed of seven Chapters and three Appendixes. Each Chapter is devoted to a key concept of **Milord II**. They are ordered to provide an incremental introduction to the language **Milord II** and its semantics.

**Chapter 2:** contains the description of the modular component of the language. It presents the syntax and the semantics of modules, generic modules and the operations between modules like refinement, contraction and expansion.

**Chapter 3:** is devoted to the approximate reasoning. We present the algebra of truth–values used in **Milord II** and the set of operations which defines a logic language of order 0. After that the extension of that algebra to an algebra of intervals of linguistic terms is introduced. It allows us to introduce imprecision in **Milord II**. We present fuzzy sets as a method to talk about sets in **Milord II**. Finally we deal with local logics and the form to find valid mappings among the different logics of modules.

**Chapter 4:** addresses to the concept of specialization of KBs. We define the specialization calculus for a multi–valued logic language. We present the definition of the inference engine that implements that calculus. Finally we introduce the concrete syntax of the deductive knowledge of **Milord II** and all the extralogical components.

**Chapter 5:** considers the local control of **Milord II** that is composed of the implicit control and the explicit one declared by the user.

**Chapter 6:** is devoted to the applications and examples that have been developed using **Milord II** system.

**Chapter 7:** summarizes the conclusions of this work.

After that we include three appendixes. Appendix A contains a complete BNF description of the syntax of the language **Milord II**. Appendix B contains the proofs of the theorems appeared in Chapter 4. And Appendix C contains complete coded examples of applications developed using **Milord II**.

Constructs of the language are introduced by means of examples of real applications developed using **Milord II** mainly from *Terap–IA* and *Spong–IA* expert systems[4]. We try to give simple descriptions of the components of the system.

[4]For the sake of simplicity examples from applications are simplified to give only an idea of the constructs introduced. A general view of these applications is presented in Chapter 6 and Appendix C.

# Chapter 2

# Modularity

Classical software engineering approves top–down design as a good programming methodology. Decomposition of a whole problem into simple subproblems allows us to have a clear gain in clarity, simplicity, complexity degree and debugability of programs. **Milord II** is a programming environment that offers all the advantages of the structured problem solving.

The experience of our group in Knowledge Based Systems (KBSs) design and development, specially using knowledge acquisition techniques (Plaza and López de Mántaras, 1989), have allowed us to detect a number of needs that can be tackled with the methodology we propose here. Among them we can emphasize: Modularity, multilanguage representation, local control, reusability, incremental development and validation. Let us briefly discuss the meaning of all these:

**Modularity.** The usual way of understanding a complex problem is to decompose it into simple subproblems using simple operations. To make a useful decomposition of problems, subproblems must have a simple and well defined interaction. The determination of the adequate nature of modules, or partial Knowledge Bases (KB's), that represent the subproblems and the definition of the combination operations of these partial KB's, are key points in the design of a language for Knowledge Engineering.

**Multilanguage representation.** The basic operations of modularization and modification of modules are independent from the underlying language used to define the bodies of the modules. This independence allows the use of different representation languages in the different modules (Harper et al., 1989). A simple example of this is the use of different multi–valued logics in each module (Sierra and Agustí, 1991).

**Local Control.** Control is a component of the problem solving task tied to the domain knowledge. Thus it must be a component of each module.

**Reusability.** In the building process of a KB it is important to be able to reuse existing partial KB's of problems solved beforehand (Chandrasekaran, 1986;

Goguen, 1986). For instance, although the diagnosis of infectious chest illness and that of chest tumors are essentially different, they could share the knowledge of an analysis of a thorax radiography. This is an example of two modules that share the same submodule. There are many examples of this. Gram analysis is a task that is independent of the type of sample we are considering. We can program gram analysis as a generic task instead of a specific gram analysis for every type of sample. So, as a requirement of our language, we need generic program units that could be instantiated, or reused, in different contexts. These are the generic modules.

**Incremental modification of KB's.** The KB building methodology is an iterating two–step process. First a prototype is build (or modified), then it is validated. Thus, it is convenient to have some safe refinement operations in the language that support this process of incremental KB building (modification). These operations have to preserve the adequacy of the KB behavior with respect to the behavior required by the expert as stated in the export interface of each module.

**Validation.** Normally KB validation is applied only to the KB considered as a whole and after it has been completely build. We think validation should be done during the KB building process in each module, i.e., in the different and successive partial KB's or modules that, conveniently combined and progressively refined, will result in the total KB. The validation should not be just a final quality control test, but it must be integrated into the building process of the system. We can use any validation method for every module that belongs to a whole ES. Thus the complexity degree of the validation task diminishes considerably.

All the above issues can be grouped taking into account the classical *cycle of software*. Modularity, local control and multilanguage representation are then tied to the development process of an ES. These are related to the decomposition of problems and the encapsulation of information. We can build each problem unit using the information, control and representation language more adapted to it. Reusability and incremental modification of KB's are related to the modification of an existing system. The techniques that help in the modification of KB's are very useful. After a first design we should validate it and modify it if necessary.

## 2.1 Introduction

All these considerations have determined the design of **Milord II**. Now we introduce some precedents of **Milord II** and the main constructs that the language provides to satisfy the above needs.

### 2.1.1 Previous Work

The construction of modular expert systems started with the work in *Milord* (Agustí et al., 1989; Sierra and Agustí, 1991). The main idea was to adapt to ES's the modularization techniques of the language ML (Harper et al., 1986), and use them to make modular the rule language *Milord*. A key feature of these techniques is that the modular language is independent of the underlying language. Similar work was previously done with functional and logic programming languages (Sannella and Wallen, 1987; O'Keefe, 1985; Miller, 1986).

We should consider two aspects of this first attempt. The first one is about the modular language of *Milord*, and the second one is about its implementation.

*Milord* modular language provided a tool based on modules and generic modules to structure the domain knowledge of an application. Notice that modularization affected only the domain knowledge (control knowledge was a global component). This sort of technique was useful for the *Milord* goals, that was: to grow down the design difficulty using a discipline of structuration, to control the possible errors, etc. The control and the declaration of the multi–valued logic for the applications remained global.

In **Milord II** we propose a more radical modularization technique (Puyol et al., 1991) in the sense that we avoid global components in the system. As explained above local control and multilanguage representation are desirable characteristics of a modular expert system. Then **Milord II** introduces local control and local multi–valued logics into the modules.

Now we come to the implementation aspect. An application written in *Milord* modular language was compiled to a *flat*[1] structure (a set of *Milord* rules, the core language of *Milord*) as described in (Agustí et al., 1989). *Milord* modularity was based in a compiler to translate a modular program to an equivalent flat one, and then using the *Milord* interpreter for the flat program. This solution was taken to keep the rule interpreter of *Milord*.

The modular language of **Milord II** is not compiled to a flat one. The proposal of **Milord II** is not just adding some syntactic modular facilities to the rule–based language but gives a semantic approach close to object oriented languages. Modules of **Milord II** have its own entity at runtime and we are able to give a semantic interpretation of the modules as specialists (Sticklen et al., 1987) in some aspect of an application. The interpreter of **Milord II** is oriented to the execution of the modules and the communication among them.

### 2.1.2 Modular System

In this Chapter we explain the modular components of **Milord II**, that is, the concepts of modules, submodules, generic modules and the refinement, expansion and contraction of modules. The internal components of modules (deductive knowledge, local logics, local control, etc) will be explained along this thesis.

Before to enter in the concrete syntax and semantics of **Milord II** modular language let us to introduce informally the main concepts by means of an ex-

[1]We consider that a structure is *flat* when it is no modular .

ample from *Bacter–IA* application (microbiological analysis for the diagnosis of pneumoniae) that will be used along this Chapter.

### Modular Hierarchy

One of our main goals is to design a language that allows us programming in the large. The normal method is to divide the problem in a set of simpler subproblems. This leads us to the notion of modules and submodules hierarchically organized, representing the modules problems, and its submodules the decomposition of these problems into subproblems. Every subproblem can be recursively decomposed to its subproblems resulting in a hierarchical structure of modules.

First we should clarify which is the notion of problem used in **Milord II**. To define a problem we must precise first what we consider is a solution for that problem, which are the useful data we need to know in order to obtain that solution and how to obtain that solution from that data. For instance, consider a very simplified problem of pneumoniae diagnosis. The solution to that problem is to find the germ causing pneumoniae. The data is relative to the patient. The solution to that problem is then to give certainty degrees to a set of concepts (in this case germs): *pneumococcus*, *haemophilus*, *staphylococcus*, and so on. A possible solution could be *pneumococcus is very possible and staphylococcus is slightly possible*. Obviously we need to know relevant data (input) about a concrete case to be able to find those solutions, for instance the parameters of a microbiological analysis of a sputum sample of the patient, or data about the kind of infection he has.

The specification of a problem then consists in identifying a set of goals to achieve, and the elements needed to solve them. Modules implement functional abstraction, in the sense that we can see a module as a blackbox, and we know which are the requirements of the module (input) to reach exported results (output). The notion of module is based on the concepts of encapsulation and information hiding. Encapsulation consists in grouping the components that are useful to reach a concrete set of goals. Information hiding is realized declaring inside a module which components are visible from outside the module. All the other components are effectively hidden. From the problem specification of the previous example we can build a module with outputs the germs and as input the necessary data to give a certainty value to the germs.

The above problem is a complex one, it may be decomposed in a set of subproblems. For instance, the problem of finding the germs causing pneumoniae can be simplified by decomposing it in four submodules: the first one is devoted to obtain a respiratory diagnosis of the patient; the second one finds the kind of infection the patient has; the third one informs about the previous treatment that has been administrated to the patient and finally the last one consist in a gram analysis of a sample of the sputum of the patient.

All these subproblems provide useful data for giving certainty values to the set of germs cited above.

### Generic Modules

We have structured the problem in subproblems and the system would be more powerful if it allows the reutilisation of these units in the case of similar subproblems. For instance, we can think on the previous problem of finding the germ causing *pneumoniae*. We have seen that the solution for the problem depends on the solutions of a set of subproblems. Remember that one of the subproblems was a gram analysis of a sample of the sputum of the patient.

However some data obtained from a gram analysis of the sputum could be obtained from different gram analysis over different samples. In this case it is not necessary to define a different problem solution for each type of analysis, it would be enough to define a generic problem solution depending on the kind of analysis. We incorporate generic modules in the language, which are modules depending on other modules as parameters (module variables or inputs) or also called parameterized modules. Then we can obtain the concrete subproblems (modules) through the instantiation of generic modules by substitution of parameters with concrete modules. In the example it is shown the instantiation of a generic problem with a concrete laboratory analysis of a sample.

### Refinement, Expansion and Contraction of Modules

In the introduction we talked about providing tools to the user to facilitate the construction of knowledge bases. One of the facilities of the modular language is that it allows us to decompose a problem in a set of modules. Furthermore we can introduce other tools to aid the construction of each module. We introduce in the language the notion of refinement, a sort of incremental programming. We can specify a module incrementally, that is, from the first version of a module to other versions of the same module that are refinements of the previous version (more detailed problem description). Then we can incrementally build more concrete versions of the module until a final version is achieved. Similarly we can say that a module is an expansion or a contraction of a previous module (we expand or contract the set of problems that the previous module was able to solve). In the example above we can say that the description of a sample of sputum is a more refined version that the general description of a sample.

In the following we introduce the syntax of modules progressively. We only describe the simplified syntactical categories as they are needed in each Section. For a complete description please consult in the Appendix A.

## 2.2 Primitive Components

Now we explain the concrete syntax of the declaration of modules that allows us to declare the concepts we have introduced informally. For that we will use the same example.

Modules are composed of a set of declarations: import and export interfaces, dictionary, submodules, rules, control, metarules, logic and so on. The declaration of the submodules settles the hierarchic structure of the module. The

declaration of submodules is identical in every aspect to the declaration of the modules.

The components of modules are the following (see the example from *Bacter–IA* for the module *Gram* in Figure 2.1):

**Interface:** The interface of a module has two components: the import and the export interface. They implement the external requirements (from the user) and the results of a module. All the facts inside a module not declared in the export interface are hidden to the outside of the module. In the current example the export interface of the module *Gram* is the set of germs cited above. In this case this module has not import interface because it does not need data from the user.

**Hierarchy:** The hierarchy of a module is a set of submodule declarations. A module has visibility over all the facts exported by its submodules. In the example the module *Gram* has four submodules (*D*, *T*, *P* and *S*). They are declared in Figures 2.3 and 2.8.

**Kernel:** The kernel allows modules to deduce the components of its export interface from the components of its import interface and those of the export interfaces of its submodules. The kernel of a module is made up of two components called *deductive knowledge* and *control knowledge*. Deductive knowledge includes the declarations of the object language which in our current implementation is a rule–based language. Control knowledge is represented by means of a meta–language which acts by reflection over the deductive knowledge and the hierarchy of the module. A module with an empty kernel can be considered to be a pure interface. In our case the main components of the code of a module are basically a set of rules and meta–rules to be interpreted by an inference engine.

The language provides three basic mechanisms of module manipulation:

1. Composition of modules through the declaration of submodules.

2. Composition of modules through operators defined by the user via generic modules definition.

3. Refinement, expansion and contraction of modules.

In this Chapter we will introduce the module declarations and the mechanisms of module manipulation mentioned above. The components of the kernel of a module as the deductive and the control declarations will be presented in Chapters 4 and 5 respectively.

```
Module Gram =
    Begin
    Module D= Respiratory_Diagnosis
    Module T= Type_of_Infection
    Module P= Previous_Treatment
    Module S= Gram_of_Sputum
    Export Pneumococcus, Haemophilus, Staphylococcus, Enterobacteria
    Deductive knowledge
        Dictionary: not defined here
        Rules:
        R001 If S/DCGP then conclude Pneumococcus is possible
        R002 If S/DCGP and D/Bact_Pneumonia
            then conclude Pneumococcus is very_possible
        R003 If S/BGN and D/Aspiration_Pn and T/Nosocomial
            then conclude Enterobacteria is quite_possible
        R004 If S/CBGN and P/Penicilin
            then conclude Haemophilus is sure
        Inference system:
        Truth values= (impossible, few_possible, sligh_possible, possible,
            quite_possible, very_possible, sure)
        Renaming
            D/False ==> impossible
            D/True ==> sure
            T/False ==> impossible
            T/True ==> sure
            P/impossible ==> impossible
            ...
        Connectives:
            Conjunction = Truth Table
                ((impossible impossible impossible impossible,
                    impossible impossible impossible)
                ...
                (impossible few_possible sligh_possible possible
                    quite_possible very_possible sure))
        end deductive
    Control knowledge
        Evaluation Type: Lazy
        ...
        end control
    end
```

Figure 2.1: Example of module declaration.

| | | |
|---|---|---|
| interface | ::= | [import] |
| | | [export] |
| import | ::= | **Import** predicateidlist |
| export | ::= | **Export** predicateidlist |
| predicateidlist | ::= | *predid* , predicateidlist \| *predid* |

Figure 2.2: Syntax of interfaces.

### 2.2.1 Interfaces of modules

Figure 2.2 contains the syntax of the interface declarations, that consist in a list of facts for each interface.

Imported facts are those facts whose values can be obtained from the user during the execution of a module. For instance, the module *Previous_Treatment* of the Figure 2.3 has the following declaration:

**Import** Prev_Treat

Imported facts like *Prev_Treat* are asked when needed in the evaluation of a module[2]. The code of a module containing an import declaration will be allowed to ask to the user for values of imported facts only. For instance the module of the previous example can only ask to the user for the value of the fact *Prev_Treat*.

Exported facts are those facts that are visible outside the module. They can be asked by the user or by other modules. For instance, the module *Previous_Treatment* of the Figure 2.3 has the following export interface declaration:

**Export** Penicilin, Tetracycline

All the exported facts like *Penicilin* either have to be deduced by the kernel of the module or have to be imported by the module (obtained from the user). In this example the facts of the export interface can be deduced by the kernel by means of the rules *R001* and *R002*. Notice that the facts of the export interface of the module *Type_of_Infection* of the Figure 2.3 are imported directly from the user (*Nocosomial* and *Extrahospitalary* are facts that belong both to the export and import interface of the module). Facts deduced and imported facts not mentioned in the export declaration of the module are hidden to the rest of the modules including the user, i.e., they cannot be used in the body of the rest of the modules. A module with no exported facts is meaningless. However we can access to the exported facts of its submodules as explained in subsection *access names* below. The code of a module containing an export declaration will provide means to answer questions about the values of the exported facts only.

[2]When and in which order the imported facts are asked to the user is determined by the type of evaluation of the module. See the Section 5.3.

```
Module Respiratory_Diagnosis =
    Begin
    Import Bact_Pneumonia,Influenz_superinf, Aspiration_Pn, Cronic_Pn
    Export Bact_Pneumonia,Influenz_superinf, Aspiration_Pn, Cronic_Pn
    Deductive knowledge
    Dictionary: not defined here
        Inference system:
        Truth values = (false, true)
        Connectives:
            Conjunction = Truth Table
                ((false false) (false true))
    End deductive
    End

Module Type_of_Infection =
    Begin
    Import Nosocomial, Extrahospitalary
    Export Nosocomial, Extrahospitalary
    Deductive knowledge
    Dictionary: not defined here
        Inference system:
        Truth values = (false, true)
        Connectives:
            Conjunction = Truth Table
                ((false false) (false true))
    End deductive
    End

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

Figure 2.3: Example of module declaration.

### 2.2.2 Modular hierarchy

Writing a **Milord II** program implies to start defining modules. The primitive syntax of module declarations is described in Figure 2.4. A module declaration (moddecl) is composed of a module identifier (*amodid*) and a body (bodyexpr). The body of a module can contain other module declarations (hierarchy). We say that the modules in the hierarchy declaration are submodules of the module that contains the declaration. For instance the module *Respiratory_Diagnosis* is a submodule of the module *Gram* (see Figure 2.1).

| | | |
|---|---|---|
| moddecl | ::= | **Module** *amodid* = bodyexpr |
| bodyexpr | ::= | pathid \| |
| | | **begin** decl **end** |
| pathid | ::= | *amodid* \| *amodid*/pathid |
| decl | ::= | [hierarchy] |
| | | [interface] |
| | | [deductive] |
| | | [control] |
| hierarchy | ::= | moddecl \| |
| | | hierarchy hierarchy |

Figure 2.4: Syntax of modules.

The language provides two types of module declarations, depending on the type of body declaration. We will use as an example the set of modules in Figures 2.1 and 2.3.

#### Encapsulated declarations

Encapsulated declarations are the most primitive module declarations. They associate a module identifier to a body. The body declaration gives a complete description of the module. The module declaration between keywords **begin** and **end** contains the hierarchy, interface and kernel declarations (deductive and control knowledge) of the module.

**Module** Gram = **Begin** ... **End**

For instance, the module *Gram* (Figure 2.1), and the modules *Respiratory_Diagnosis*, *Type_of_Infection* and *Previous_Treatment* (Figure 2.3) are examples of encapsulated declarations.

#### Declarations by reference

Module identifiers are used to refer to modules[3]. For instance in Figure 2.1, the hierarchy declaration in the module *Gram* references the module identifier

[3] The referred modules may not have been created in the moment when their names are used, expediting thus a top down design. In the following we consider for simplicity that symbols

*Respiratory_Diagnosis* with the following declaration:

**Module** D = Respiratory_Diagnosis

In this case $D$ is the internal name of the module *Respiratory_Diagnosis* used in the module that contains this declaration (*Gram*).

**Access names**

With the declarations contained in the above examples we obtain the module *Gram* which has four submodules: *Respiratory_Diagnosis*, *Type_of_Infection*, *Previous_Treatment* and *Gram_of_Sputum*. The internal names $D$, $T$, $P$ and $S$ correspond to these submodules. They are used to reference the facts exported by each of this submodules.

Paths of module names (pathid) indicate how to access a module in the hierarchy of modules. A path to a module is composed by module names separated by a *slash* character "/". For instance the path *Gram/D* references the module *Previous_Treatment*.

The access to exported facts of modules is composed of a path to a module, the *slash* character and the name of the exported fact. For instance, to access the exported fact *Extrahospitalary* of module *Type_of_Infection* we can use the following equivalent names[4].

Type_of_Infection/Extrahospitalary
≡ Gram/T/Extrahospitalary

Given a module we can access to the exported facts of that module and to the exported facts of its submodules, using the adecuate paths.

We can use encapsulated declarations or declarations by reference depending on the structure we want to give to our problem. If we use only encapsulated declarations we will produce a structure with only a top level module. All the other modules have to be accessed by means of paths of module names.

### 2.2.3 Semantics of modules

At this point it is interesting to give a formal description of the modular environment of **Milord II**. After that we will extend this description with new elements. We will keep the same example of Figure 2.1.

A program is a table $P$ from system module identifiers $Id_{\mathcal{M}}$ to the set of modules $\mathcal{M}$:

$$P = Id_{\mathcal{M}} \rightarrow \mathcal{M}$$

The set of system module identifiers $Id_{\mathcal{M}}$ is a set of internal names to distinguish the different modules (we will use names like $mod_1, \ldots, mod_i$ for the set

exist when they are referenciated, that is, there are already all the encapsulated declarations that are required.

[4]At the moment we consider that all the modules are visible. In the following Sections we will see cases where not all the possible paths are allowed. For instance, the submodule $T$ could be hidden outside the module *Gram*.

$M$ and $\top$ for the top module). The set of modules $\mathcal{M} = M \cup \{\top\}$ is composed by the set of modules $M$ that are created from all the encapsulated module declarations (**begin** ... **end**), plus a virtual module named $\top$ (*top module*).

In our example (Figures 2.1 and 2.3) there are five declarations of this type. They create a set of six module identifiers.

$$Id_{\mathcal{M}} = \{\top, mod_1, mod_2, mod_3, mod_4, mod_5\}$$

Each module $m \in \mathcal{M}$ is composed of the hierarchy ($H$), export ($E$), import ($I$) and kernel ($K$) components. The top module only contains the top level hierarchy of modules (export[5], import and kernel components are empty).

$$\mathcal{M} = H \times E \times I \times K$$

In the following to simplify the notation we will use $H_m$ as the hierarchy component of a module $m \in \mathcal{M}$[6]. Similarly for the export, import and kernel components ($E_m$, $I_m$ and $K_m$ respectively). The hierarchy of a module $m \in \mathcal{M}$ is a function $H_m$ from local submodule identifiers of the module $m$, $\mathcal{I}_m$, to identifiers of the modules $M$ ($Id_M$) and a visibility module qualifier (visible, hidden or open[7]).

$$H_m : \mathcal{I}_m \rightarrow Id_M \times \{visible, hidden, open\}$$

Each module has its own function that points to its submodules. Following the same example, we will show the hierarchy components of those modules (see the Figure 2.5). The top level module contains all the modules declared by means of encapsulated declarations at top level:

$$\mathcal{I}_\top = \{Gram, Respiratory_Diagnosis, Type_of_Infection, \\ Previous_Treatment, Gram_of_Sputum\}$$

$$\begin{aligned} &H_\top(Gram) = (mod_1, visible) \\ &H_\top(Respiratory_Diagnosis) = (mod_2, visible) \\ &H_\top(Type_of_Infection) = (mod_3, visible) \\ &H_\top(Previous_Treatment) = (mod_4, visible) \\ &H_\top(Gram_of_Sputum) = (mod_5, visible) \end{aligned}$$

The module *Gram* has four submodules declared by reference:

$$\mathcal{I}_{mod_1} = \{D, T, P, S\}$$

$$\begin{aligned} &H_{mod_1}(D) = (mod_2, visible) \\ &H_{mod_1}(T) = (mod_3, visible) \\ &H_{mod_1}(P) = (mod_4, visible) \\ &H_{mod_1}(S) = (mod_5, visible) \end{aligned}$$

[5]As noticed above a module with empty export interface has no sense. In that case the top module is a special module and we are only interested in accessing the exported facts of its submodules.

[6]For instance the hierarchy of a module $m$ could be expressed as $H_m = first(P(m))$, the first component of the tuple corresponding to identifier $m$.

[7]In Section 2.5.1 we will see *open* declarations of submodules. Now we consider all the modules are visible.

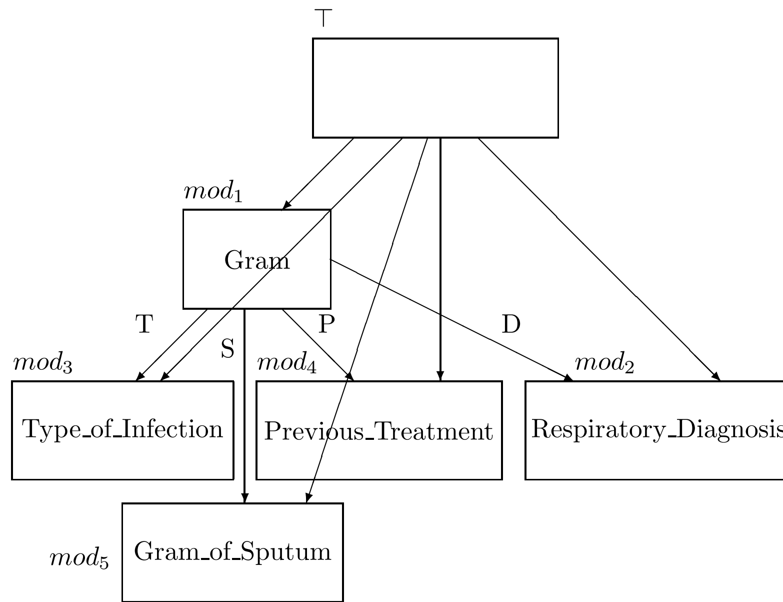


Figure 2.5: Hierarchy example.

After the definition of the semantical structure of a program and given the functions that allow us to access the elements of that structure, we define the main function that is used to execute a program.

Let us introduce the function to query an ES. **Query** is a command that given a path name (a question prefixed by a path) returns an answer to that question. It will be the main function of the system that allows us to start the execution of an ES. The syntactical form is the following:

**Query** query
query ::= factid | moduleid/query

For instance, with the set of modules introduced above we can start the execution with the following query from the user:

**Query**(Gram/P/Peniciline)

The functions $Query_m(x)$ are internal queries to the modules, they represent a query, of fact $x$ to the module $m$. Its definition is the following:

$$Query_m : query \rightarrow Answer$$

$$Query_m(moduleid/query) = \begin{cases} Query_x(query) & \exists x, H_m(moduleid) = \\ & (x, visible) \\ error & \text{otherwise} \end{cases}$$

$$Query_m(factid) = \begin{cases} deduce_m(factid) & \text{if } factid \in E_m \\ error & \text{otherwise} \end{cases}$$

Notice that when we query a module for a fact belonging to a submodule of that module, this module has to be visible. The execution of a query always starts on the top module. For instance, consider the command above, where the **Milord II** interpreter ($\mathcal{INT}$) makes a query to the top module of the current ES.

$$\begin{array}{l} \mathcal{INT}[\![\textbf{Query}(\text{Gram/P/Peniciline})]\!] \\ Query_{\top}(Gram/P/Peniciline) = \\ Query_{H_{\top}(Gram)}(P/Peniciline) = \\ Query_{mod_1}(P/Peniciline) = \\ Query_{H_{mod_1}(P)}(Peniciline) = \\ Query_{mod_4}(Peniciline) = deduce_{mod_4}(Peniciline) \end{array}$$

Finally the interpreter starts the deduction in the module $mod_4$ (expressed by $deduce_{mod_4}$) in order to give a value to the fact *Peniciline*. We will return to this point along the thesis. We have introduced the modular structure of a program from the semantic point of view. In the following we will extent this semantical description adding new components that we will progressively introduce.

## 2.3 Generic modules

The instantiation of a generic module is considered as a module declaration. The definition of generic modules opens to the user the possibility of defining specific operations of composition of modules. Generic modules are then operations (or functions) on modules. This standard technique to define generic modules is the one to define functions, that is, it consists of isolating a piece of program, or module, from its context and then abstracts it by specifying:

1. Those modules upon which the abstracted module may depend (requirements or parameters of the generic module).
2. The contribution of the abstracted module to the rest of the program (results or export interface of the generic module).

As we said the obvious example of this technique is functional programming, where such abstractions (functions) form the basic program units. The functional body defines how to compute the output (results) in terms of the input (requirements). The method for building large KB systems consists of applying generic modules to previously built particular modules.

An example of definition of generic modules is presented in figure 2.6. Remember the example used until now. The module *Gram* (Figure 2.1) has four submodules (Figures 2.3 and 2.8). The submodule $S$ corresponds to the gram analysis of sputum. Now the generic module *Global_Gram* (Figure 2.6) represents a general gram analysis over different samples. *Global_Gram* can now be

applied to different modules to produce new modules analyzing different types of samples. For instance, an equivalent of module *Gram* (named *New_Gram*) can be obtained by instantiating the generic module *Global_Gram* with the module *Gram_of_Sputum*.

Keeping the common parts in a generic module we can save code and time and make the code much more understandable. Finally when a module is needed to make the gram analysis of a sputum sample, it is only necessary to put both modules together by a generic module application, for instance *Global_Gram(Gram_of_Sputum)*.

In Figure 2.7 we extend the syntax of modules declaration of Figure 2.4 adding the generic modules declarations.

The declaration of a generic module is like a normal module except that in the hierarchy component of that module there are submodules with local names the parameters of the generic module. These submodule names point nothing until the instantiation of the generic module. Then the instantiation of a generic module consists in:

- Making a copy of the generic module.
- Solving the hierarchy component of that copy.

For instance, we can add to our current system the generic module *Global_Gram*, as the new module $mod_6$. The hierarchy component of that module is:

$H_{mod_6}(D) = (mod_2, visible)$
$H_{mod_6}(T) = (mod_3, visible)$
$H_{mod_6}(P) = (mod_4, visible)$
$H_{mod_6}(X) = \emptyset$

To compute *Global_Gram (Gram_of_Sputum)* we make then a copy of that module ($mod_7$) and solve the hierarchy component linking the submodule $X$ with the module *Gram_of_Sputum*. This is the module *New_Gram*. It is equivalent to the previous one (*Gram*), except that the submodule $X$ (*Gram_of_Sputum*) is hidden outside *Global_Gram*.

$H_{mod_7}(D) = (mod_2, visible)$
$H_{mod_7}(T) = (mod_3, visible)$
$H_{mod_7}(P) = (mod_4, visible)$
$H_{mod_7}(X) = (mod_5, hidden)$

The parameters of a generic module are considered as submodules. If we want to refer to exported facts of these submodules we must build a path with the parameters names (for instance in the rules of the module *Global_Gram* we have *X/CBGN*).

The parameters of a generic module are submodules hidden outside the generic module. For instance, notice that the submodule $X$ is hidden outside the new module *New_Gram*. The following query is then wrong because the submodule $X$ is hidden ($H_{mod_7}(X) = (mod_5, hidden)$):

```
Module Global_Gram (X) =
    Begin
    Module D= Respiratory_Diagnosis
    Module T= Type_of_Infection
    Module P= Previous_Treatment
    Export Pneumococcus, Haemophilus, Enterobacteria
    Deductive knowledge
        Dictionary: not defined here
        Rules:
        R001 If X/DCGP then conclude Pneumococcus is possible
        R002 If X/DCGP and D/Bact_Pneumonia
            then conclude Pneumococcus is very_possible
        R003 If X/BGN and D/Aspiration_Pneumonia and T/Nosocomial
            then conclude Enterobacteria is quite_possible
        R004 If X/CBGN and P/Penicilin then conclude Haemophilus is sure
        Inference system:
        Truth values= (impossible, few_possible, sligh_possible, possible,
            quite_possible, very_possible, sure)
        Renaming =
            D/False ==> impossible
            D/True ==> sure
            T/False ==> impossible
            T/True ==> sure
            P/impossible ==> impossible
            P/sure ==> sure
            X/false ==> impossible
            X/unlikely ==> [impossible, possible]
            X/may_be ==> possible
            X/likely ==> [possible, sure]
            X/true ==> sure
        Connectives:
            Conjunction: the same table defined in Figure 2.1.
        end deductive
    end

Module New_Gram = Global_Gram(gram_of_sputum)
```

Figure 2.6: Example of generic module definition and application.

| | | |
|---|---|---|
| moddecl | ::= | **Module** modbind |
| modbind | ::= | *amodid* [([paramlist])] [≡ bodyexpr] |
| bodyexpr | ::= | *pathid*[([iparamlist])] \| |
| | | **begin** decl **end** |
| paramlist | ::= | *amodid* \| paramlist ; paramlist |
| iparamlist | ::= | bodyexpr \| iparamlist ; iparamlist |

Figure 2.7: Syntax of generic modules.

**Module** Gram_of_Sputum =
  **Begin**
  **Import** Sputum_Clas, Sputum_Gram
  **Export** End, Gram_yes, DCGP, CGPC, CGPR, BGN, CBGN
  **Deductive knowledge**
  **Dictionary:** *not defined here*
    **Rules :**
    R001 **If** Sputum_clas=(Grup_1,Grup_2,Grup_3)
        **then conclude** Sputum_ok **is** true
    R002 **If** Sputum_clas=(Grup_4,Grup_5,Grup_6)
        **then conclude** Sputum_not_ok **is** true
    R003 **If** Sputum_ok **then conclude** Gram_yes **is** true
    R004 **If** Sputum_not_ok **then conclude** End **is** true
    R005 **If** Sputum_Gram=(DCGP_MC) **then conclude** DCGP **is** true
    R006 **If** Sputum_Gram=(DCGP_MC) **then conclude** CGPC **is** unlikely
    R007 **If** Sputum_Gram=(CGPC_MC) **then conclude** CGPC **is** likely
    R008 **If** Sputum_Gram=(CGPC_MC) **then conclude** DCGP **is** unlikely
    R009 **If** Sputum_Gram=(CGPR_MC) **then conclude** CGPR **is** true
    R010 **If** Sputum_Gram=(BGN_MC) **then conclude** BGN **is** likely
    R011 **If** Sputum_Gram=(CBGN_MC) **then conclude** CBGN **is** likely
    **Inference system:**
    **Truth values =** (false, unlikely, may_be, likely, true)
    **Connectives:** ...
  **End deductive**
  **End**

Figure 2.8: Example of module used as parameter of a generic module.

$\mathcal{INT}[\![\textbf{Query}(\text{New_Gram/X/CBGN})]\!]$
$Query_{\top}(New_Gram/X/CBGN) =$
$Query_{H_{\top}(New_Gram)}(X/CBGN) =$
$Query_{mod_7}(X/CBGN) = error$

A generic module can use the exported facts of its module parameters, so the instantiation of a parameter can not be made with any module, but only those exporting the needed facts. For instance the module *Gram_of_Sputum* exports the facts needed in the rules of the generic module *Global_Gram*. In the following we will see how to make save the instantiations of generic modules by means of declaring the parameters as refinements of another module (like giving a type to the parameters).

We want to support the process of incremental KB building by means of generic modules. So whenever the definition of a generic module changes, these changes must be reflected in the rest of the program. The way to do it is just to repeat the module applications that refer to the modified module. This re–linking process can be automatised by the compiler, so that the user gets rid of this task.

## 2.4 Refinement, Expansion and Contraction

Before explaining these operations on modules we need to know which are the main components of the kernel of modules. It is not our purpose here to explain in detail the kernel of modules. We only give a short description of the main semantical components of the kernel which are needed to understand the explanations of the current Chapter.

The kernel of modules is composed of the deductive and the control knowledge (see the Figure 2.9) . Deductive knowledge is composed of the definition of facts (dictionary), the rules and the definition of the local logic (inference system). At the moment we consider the kernel of a module composed of the following set[8]: dictionary, rules, logic and control.

The kernel of a module $m \in M$ is composed by the dictionary ($D_m$), the rules ($R_m$), the logic ($L_m$) and the control ($C_m$).

$$K = D \times R \times L \times C$$

So far we have described modules and generic modules. These declarations are enough to write programs. Now we introduce the operations that allow us to deal with incremental programming in **Milord II**. These operations are the refinement, expansion and contraction of modules.

Top–down programming methodology is related to an incremental specification of problems. We have seen that modular decomposition of problems is a useful technique to simplify the programming task. We can think in a program

[8]The complete explanation of the deductive and control knowledge will be done in Chapters 4 and 5 respectively. Here we only explain what is needed to understand the concept of inheritance which is closely related to modularity and affects mainly the kernel of modules.

```
Deductive knowledge
          Dictionary:  ...
          Rules:  ...
          Inference system:  ...
end deductive
Control knowledge
        Evaluation Type:  ...
        Truth Threshold:  ...
        Deductive Control:  ...
        Structural Control:  ...
end control
```

Figure 2.9: Kernel declaration scheme.

as a hierarchy of modules, but we are interested in including in the language a set of operations for assisting experts in the process of development, from the first prototype to the final version of the program.

The task of incremental programming consists in writing a first prototype, then a second one, and so on until a final version is achieved. This imposes two requirements to our system. The first one in that the partial specifications of modules must be executable to test them. The second one is that we need to define a set of operations for overseeing this process of incremental building of programs.

If you observe the concrete syntax of modules in Appendix A you can notice that many components of modules are optional. Semantically any module declaration has sense, for instance, we can declare modules without control component, or without import interface, etc. In some cases we do not declare some components of modules because we want to define them incrementally. For instance, if we execute a module which contains only an export interface, it will answer to all the questions *unknown*.

When we program a new version of a previous program we are interested in checking and declaring what is the relation with the old version. In **Milord II** this relation can be a refinement, contraction or expansion of the previous one. **Milord II** provides these set of operations at modular level, then we say that a module is a refinement of another one when the set of accessible[9] facts is the same that the previous one and these facts can be obtained with more or equal precise values[10]. When we expand the accessible facts in the next version or reduce them then we talk about expansion and contraction operations respectively.

In Figure 2.10 we extend the previous syntactical declarations of modules with these new module operations. The symbols ":", ">" and "<" stand respectively for the module refinement, expansion and contraction operations. They

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

[10]Precision is a topic that will be explained in Section 3.2.

can be used in all the module declaration including the parameter declaration of generic modules.

| | | |
|---|---|---|
| moddecl | ::= | **Module** modbind |
| modbind | ::= | *amodid* [([paramlist])] [modoper modexpr] [≡ modexpr] |
| modexpr | ::= | bodyexpr \| bodyexpr modoper modexpr |
| bodyexpr | ::= | *pathid*[([iparamlist])] \| **begin** decl **end** |
| paramlist | ::= | *amodid* modoper modexpr \| paramlist ; paramlist |
| iparamlist | ::= | modexpr \| iparamlist ; iparamlist |
| modoper | ::= | : \| $\geq$ \| $\leq$ |

Figure 2.10: Syntax of refinement, contraction and expansion.

All these modular operations are based in three functions: the enrichement verification, the inheritance and the information hiding. Now we explain the refinement of modules that is the operation that uses these three functions. The other operations are modifications of this refinement operation.

### 2.4.1 Refinement

When we design a module the first decision is which is the set of goals of that module. These goals are represented by the set of accessible facts of that module, those of its export interface and those of the export interfaces of its submodules. When we design by refinement a new version of that module we must maintain the same set of goals. Furthermore we must guarantee that these goals will be obtained with better or equal precision in the new version.

Consider the example of Figure 2.11. The module *Sample* only contains an export interface. The expression **Module** *Gram_of_Sputum : Sample* = **begin** ... **end** declares[11] that the module *Gram_of_Sputum* is a refinement of the module *Sample*. This is the idea of incremental programming, all the modules that are refinements of the module *Sample* (they represent samples) must keep the same export interface. The module *Gram_of_Sputum* is a refinement of the module *Sample* because it has the same export interface and other components (for instance, the deductive one) that allows the module to obtain better results for the exported facts (different from *unknown* as was the case for the module *Sample* because it has not deductive component).

This is specially useful when we declare generic modules. Remember that the instantiation of a generic module implies to assign submodules to the generic module. The resultant module should use the exported facts of the submodules

[11] There is an equivalent notation for it: **Module** *Gram_of_Sputum* = **begin** ... **end** *: Sample*.

```
Module Sample =
    Begin
        Export End, Gram_yes, DCGP, CGPC, CGPR, BGN, CBGN
    End

Module Gram_of_Sputum : Sample =
    Begin
        Export End, Gram_yes, DCGP, CGPC, CGPR, BGN, CBGN
    Deductive knowledge
    Dictionary: not defined here
        Rules :
        R001 If Sputum_clas=(Grup_1, Grup_2, Grup_3)
                then conclude Sputum_ok is true
    ...
    End
```

Figure 2.11: Example of generic module definition and application.

assigned. It is obvious that not all the modules can be used to instantiate a generic module. For instance, we can modify the previous declaration of the generic module *Global_Gram* as following:

**Module** Global_Gram (X : Sample) =
**Begin** the same declarations than in Figure 2.6 **End**

This kind of declaration assure us that the modules used to instantiate the generic module *Global_Gram* are refinements of the module *Sample* having the same export interface (for instance, we can assure that these module will export the fact *X/CBGN* needed in the evaluation of the module).

We obtain a new module from two modules by means of a refinement operation. We obtain the module *Gram_of_Sputum* refining the second declaration of Figure 2.11 by the module *Sample*. A refinement operation is composed by three operations: the enrichement verification, the inheritance and the information hiding.

We will explain these components by considering the following equivalent declarations:

**Module** M = M_1 : M_2 $\equiv$ **Module** M : M_2 = M_1

In the following consider that the internal identifier of the modules M_1 and M_2 are $mod_1$ and $mod_2$ respectively. The internal identifier of the new module M is $mod$. For instance if we consider that all the declarations are at top level, then: $H_\top(\text{M}) = mod$, $H_\top(\text{M_1}) = mod_1$ and $H_\top(\text{M_2}) = mod_2$.

### Enrichement Verification

Given a refinement step it is necessary to verify the enrichement of information. We take the following definition of enrichement between two modules.

**Definition 2.1 (Enrichement)** *We say that the module M_1 is an enrichement of the module M_2, if and only if:*

1. $E_{mod_2} \subseteq E_{mod_1}$
2. $\mathcal{I}_{mod_2} \subseteq \mathcal{I}_{mod_1}$ *and* $\forall x \in \mathcal{I}_{mod_1} \cap \mathcal{I}_{mod_2}$. $H_{mod_1}(x)$ *is an enrichement of* $H_{mod_2}(x)$ *or* $H_{mod_1}(x) = \emptyset$
3. $\forall x \in E_{mod_1} \cap E_{mod_2}$. $V(x, mod_1)$ *is more precise than* $V(x, mod_2)$.
4. $L_{mod_1} = L_{mod_2}$ *or* $L_{mod_1} = \emptyset$.

That means that the module M_1 can extend the export interface and the submodules of M_2. When a submodule is declared in both modules M_1 and M_2, they must preserve the enrichement relation. Finally we must obtain more precise values for the facts that belong to both export interfaces (precision will be the topic of Section 3.2, at the moment we can consider better values).

### Inheritance

When we declare a module as a refinement of another one we can maintain several components of the last version, to avoid the expert write twice the common components. Copy inheritance acts over submodules, facts defined in the dictionary and the local logic of the module.

When we declare the new version of a module by refinement we could declare new submodules and we must declare the same submodules that in the previous version (remember that $\mathcal{I}_{mod_2} \subseteq \mathcal{I}_{mod_1}$). In these declarations we could omit the body of some of them. That means that the module $M$ will inherit the bodies of the submodules of *M_2* that not are present in the declaration of *M_1*. In the case of declaring a body in *M_1* for a module declared yet in *M_2* then a refinement relation is performed between the two submodules.

For all the submodules $x$ that are in $\mathcal{I}_{mod_1} \cap \mathcal{I}_{mod_2}$, the submodules of the resultant module are:

$$H_{mod}(I) = \begin{cases} H_{mod_2}(I) & H_{mod_1}(I) = \emptyset \\ & \\ (first(H_{mod_1}(I)) : first(H_{mod_2}(I)) & \\ , second(H_{mod_2}(I))) & \text{otherwise} \end{cases}$$

The refinement operation makes a copy of the non modified elements of the dictionaries.

$$D_{mod} = \{x | x \in D_{mod_1} \text{ or } x \in D_{mod_2} \text{ and } x \notin D_{mod_1}\}$$

In the case of the logic, the module inherits the logic of module $M_2$ if that module contains a logic declaration.

$$L_{mod} = \begin{cases} L_{mod_2} & \text{if } L_{mod_1} = \emptyset \\ L_{mod_1} & \text{if } L_{mod_1} \neq \emptyset \end{cases}$$

**Information Hiding**

The first component of the refinement operation was the enrichement verification. One of the conditions to obtain a refinement relation between two modules is that the accessible facts of both modules must be the same. Then after checking the enrichement of information we must hide the new accessible facts of the refined module if any.

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

Until now we have considered that all modules are visible except for the parameters of generic modules. Here we introduce the concept of *hidden* submodules. When a module contains a submodule that is *hidden*, this submodule can not be referenced outside the module. The new module can reference the facts in the export interface of all its submodules, but from outside the module we can not access the hidden submodules.

Finally the components of the module $M$, after checking the enrichement of information, considering inheritance and information hiding will be the following:

- $E_{mod} = E_{mod_2}$
- $I_{mod} = I_{mod_1}$
- $\mathcal{I}_{mod} = \mathcal{I}_{mod_1} \cup \mathcal{I}_{mod_2}$
  $H_{mod}(I) =$
  $$\begin{cases} H_{mod_2}(I) & \text{If } H_{mod_1}(I) = \emptyset \\ & \text{and } I \in \mathcal{I}_{mod_1} \cap \mathcal{I}_{mod_2} \\ (first(H_{mod_1}(I)) : first(H_{mod_2}(I)) & I \in \mathcal{I}_{mod_1} \cap \mathcal{I}_{mod_2} \\ , second(H_{mod_2}(I))) & \\ (first(H_{mod_1}(I)), hidden) & I \in \mathcal{I}_{mod_1} - \mathcal{I}_{mod_2} \end{cases}$$
- $D_{mod} = \{x | x \in D_{mod_1} \text{ or } x \in D_{mod_2} \text{ and } x \notin D_{mod_1}\}$
- $L_{mod} = \begin{cases} L_{mod_2} & \text{if } L_{mod_1} = \emptyset \\ L_{mod_1} & \text{if } L_{mod_1} \neq \emptyset \end{cases}$
- $R_{mod} = R_{mod_1}$
- $C_{mod} = C_{mod_1}$

We have a small example of the refinement operation in Figure 2.12 and 2.13. Consider the first version of the program in Figure 2.12. It is composed of three modules that only contain submodule declarations and the interfaces. The module *Global_Culture_S* has two submodules named *Gram_S* and *Respiratory_Diagnosis*. The module *Gram_S* has only the submodule *Respiratory_Diagnosis*.

```
Module Global_Culture_S =
    Begin
    Module G = Gram_S
    Module D = Respiratory_diagnosis
    Export Pneumococcus_isolation, Haemophilus_isolation,
        Staphylococcus_isolation, No_microorganism_isolation, antibiogram
    Import Multiresistant_microorganism
    End

Module Gram_S =
    Begin
    Module D= Respiratory_Diagnosis
    Import Nosocomial, Extrahospitalary, Prev_Treat, Sputum_clas,
        Sputum_Gram
    Export Pneumococcus, Haemophilus, Staphylococcus, Enterobacteria
    End

Module Respiratory_Diagnosis =
    Begin
    Import Bact_Pneumonia,Influenz_superinf, Aspiration_Pn, Cronic_Pn
    Export Bact_Pneumonia,Influenz_superinf, Aspiration_Pn, Cronic_Pn
    End
```

Figure 2.12: Example of module refinement.

In the next version of the program in Figure 2.13 we can see how the module *Global_Culture* is defined as a refinement of the previous module *Global_Culture_S*. It is easy to see that the module *Global_Culture* is a refinement of the module *Global_Culture_S* because they have the same export interfaces and the same submodules. Then the enrichement verification test success and there is no information hiding. The only operation is the inheritance of components, in particular the bodies of the submodules $D$ and $G$.

The declaration of the module *Gram* is a little different because this module contains more submodule declarations than module *Gram_S*.

$$\mathcal{I}_{H_\top(Gram_S)} = \{D\}$$

$$\mathcal{I}_{H_\top(Gram)} = \{D, T, P, S\}$$

Then information hiding acts over the submodule structure and the new modules *Type_of_Infection*, *Previous_Treatment* and *Gram_of_Sputum* become hidden

```
Module Global_Culture : Global_Culture_S =
    Begin
    Module G
    Module D
    Export Pneumococcus_isolation, Haemophilus_isolation,
        Staphylococcus_isolation, No_microorganism_isolation, antibiogram
    Import Multiresistant_microorganism
    ...
    End

Module Gram : Gram_S =
    Begin
    Module D
    Module T= Type_of_Infection
    Module P= Previous_Treatment
    Module S= Gram_of_Sputum
    Export Pneumococcus, Haemophilus, Staphylococcus, Enterobacteria
    ...
    End

Module Type_of_Infection =
    Begin
    Import Nosocomial, Extrahospitalary
    Export Nosocomial, Extrahospitalary
    ...
    End

Module Previous_Treatment =
    Begin
    Import Prev_Treat
    Export Penicilin, Tetracycline
    ...
    End
```

Figure 2.13: Example of module refinement.

from outside the module *Gram*. Notice that the import interface of that module has changed because the facts imported by the module *Gram* are now imported by its submodules.

Finally we can see in Figure 2.14 the modular structure of the program. The modules *Type_of_Infection*, *Gram_of_Sputum* and *Previous_Treatment* are hidden into module *Gram*. For instance, the paths

*Global_Culture/Gram/Type_of_Infection/Nosocomial*
*Global_Culture/Gram/Previous_Treatment/Penicilin*

are incorrect, but the paths

*Global_Culture/Respiratory_Diagnosis/Bact_Pneumonia*
*Global_Culture/Gram/Respiratory_Diagnosis/Bact_Pneumonia*

are correct and equivalent. Another equivalent modular structure could be to declare the submodule *D* of module *Global_Culture_S* as:

**Module** D = Gram_S/D

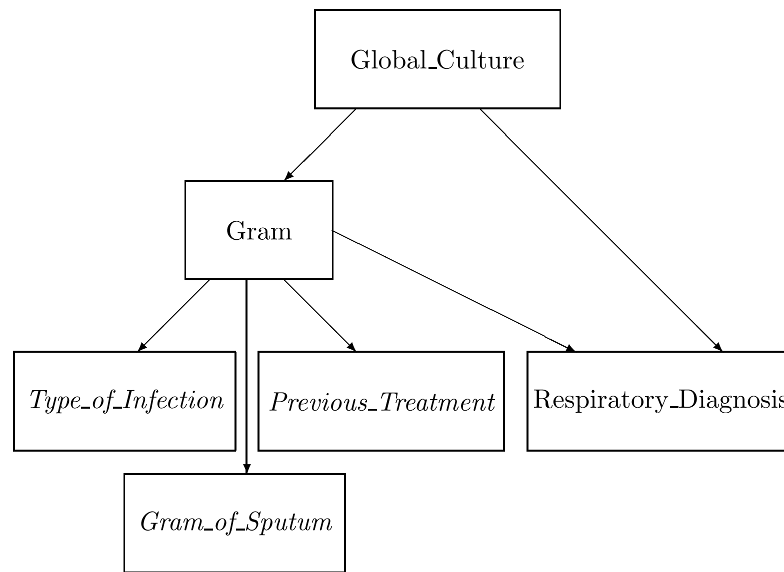


Figure 2.14: Visibility example (hidden modules are written in italic).

### 2.4.2 Expansion and Contraction

Expansion and contraction operations are based on the refinement one. Expansion allows to build modules that are an expansion of a previous version. We

can extend the set of accessible facts or add submodules to the previous version. As in the refinement case to expand modules we test that the new module is an enrichement of the previous one. Inheritance of components is performed as in the refinement operation, but information hiding is not applied because we allows the expert to program an expansion of a previous module. Consider the following declaration:

**Module** M = M_1 > M_2

The components of a module *M* created by means of the above declaration are the following (we represent only the components that are different of a refinement operation):

- $E_{mod} = E_{mod_1}$
- $\mathcal{I}_{mod} = \mathcal{I}_{mod_1} \cup \mathcal{I}_{mod_2}$
  $H_{mod}(I) =$
  $$\begin{cases} H_{mod_2}(I) & \text{if } H_{mod_1}(I) = \emptyset \\ & \text{and } I \in \mathcal{I}_{mod_1} \cap \mathcal{I}_{mod_2} \\ (first(H_{mod_1}(I)) > first(H_{mod_2}(I)) & I \in \mathcal{I}_{mod_1} \cap \mathcal{I}_{mod_2} \\ , second(H_{mod_2}(I))) & \\ H_{mod_1}(I) & I \in \mathcal{I}_{mod_1} - \mathcal{I}_{mod_2} \end{cases}$$

Contraction only test an inverse enrichement verification. Given the declaration **Module** *M = M_1 < M_2* we only test that *M_2:M_1* holds. Inheritance and information hiding are not applied.

## 2.5 Special declarations

We complete in this Section the set of module declarations allowed in **Milord II**. *Open* and *Inherit* submodule declarations are only programming facilities and they do not belong to the primitives of the modular language. *Dynamic modules* is an important characteristic of **Milord II** that allows us to execute submodule declarations at run time.

One form of the composition of modules is achieved by mean of the declaration of submodules. This declarations defines the hierarchical component of a module. The complete syntax of the hierarchy component of modules is given in Figure 2.15.

### 2.5.1 Inherit and Open

When we use *by reference* submodule declarations we usually declare a local name for the submodule. If we want to preserve the previous name of the submodule we can use the following submodule declaration:

| | | |
|---|---|---|
| hierarchy | ::= | moddecl \| |
| | | **Inherit** pathid \| |
| | | **Open** bodyexpr \| |
| | | **Sharing** patheq \| |
| | | hierarchy hierarchy |

Figure 2.15: Syntax of submodule declarations.

**Inherit** B ≡ **Module** B = B

Obviously this type of declaration can only be used in *by reference* declarations of modules.

The last type of submodule declaration is a special kind of submodule declaration, the notion of a submodule as *open*.

**Open** B

In this case it is not necessary to use paths to access the facts exported by the submodule *B*. All the facts exported by the submodule belongs to the module. It is valid with *encapsulated* or *by reference* declarations.

It is easy to see that name clashes can occur when we use declarations of type *open*. If a module has more than one opened submodule, the names of the exported facts must be different[12].

The submodules of the *open* module are visible directly without the path to the *open* module. Consider the following abstract example in Figure 2.16. We can access to the facts *C/c*, *C/a* and *C/B/b*. The paths *C/A/a* and *C/A/B/b* are not valid.

**Module** A =
    **Begin**
    **Module** B = **Begin** Export b ... **End**
    **Export** a
    ... **End**

**Module** C =
    **Begin**
    **Open** A
    **Export** c
    ... **End**

Figure 2.16: Example of *open* module.

[12]**Milord II** Compiler (Arcos, 1992) detects all these conflicts.

### 2.5.2 Sharing

Considering that **Milord II** allow us to define applications incrementally, in some cases we are interested in giving to the compiler information about which modules will be finally the same module. These declarations assure that in the building process the compiler will detect the violations of this previous declarations. Syntax declaration of sharing is given in Figure 2.17.

| | | |
|---|---|---|
| sharing | ::= | **Sharing** patheq |
| patheq | ::= | pathid ≡ patheq \| pathid ≡ pathid \| |
| | | patheq ; patheq |

Figure 2.17: Syntax of sharing.

For instance in the current example we can declare:

**Sharing** Respiratory_Diagnosis = Gram/D

This declaration means that the module *Respiratory_Diagnosis* has to be the same module that the submodule $D$ of the module *Gram*.

Sharing declaration can be used at top level or in the hierarchy declarations of modules. In the last case local names are used. The last declaration implies that:

$$H_{\top}(Respiratory_Diagnosis) = H_{\top}(Gram/D)$$

Sharing declaration can also be used in generic module declarations. In the declarations above sharing was only to give the information that some modules will be the same. In the case of generic modules sharing declarations affect the future instantiations of those generic modules. For instance, consider the following declaration:

**Module** G(X : X_S; Y : Y_S; **Sharing** X/A = Y/B) =
**Begin** ... **End**

This declaration means that when we instantiate the generic module $G$ with two modules, the first module $X$ must contain a submodule named $A$ and the second one $Y$ a submodule named $B$, and these two submodules must be the same.

### 2.5.3 Dynamic Modules

**Milord II** deals with dynamic modules by means of dynamic links among modules at run time. This characteristic allows us to implement powerful particular control strategies.

Despite the creation of dynamic modules belongs to the control knowledge of modules (see Section 5.4.4) we introduce briefly those declarations.

We can use as conclusions of metarules module declarations only composed *by reference* declarations. For instance, we can write in the control knowledge of a module a metarule which conclusion is the module declaration *Open A*. This means that when this metarule is fired then the module that contains it will stablish a dynamic link with the module A. Now this module has a new submodule.

A more interesting use of dynamic modules is the dynamic instantiation of generic modules. This technique has been useful in the development of *Spong–IA* application. It allows to build dynamically a hierarchy of modules representing the taxonomic tree for the identification of sponges. A detailed explanation can be found in (Domingo, 1995).

## 2.6 Conclusions

All the ES programming activity with **Milord II** language is based on modules. Modules are the primitive components of the language. The applications programmed with **Milord II** start by structuring the whole problem in a hierarchy of modules. It is a language adapted to the programming in the large, that is, to program real applications.

**Milord II** has not global components in the system. Each module contains a complete ES specialized in a part of the whole application. Modules has its own deductive knowledge (dictionary, rules and so on), its own local logic (particular multi–valued logic used to cope the concrete subproblem) and the local control. Modules has well defined interfaces to interact with the user and the other modules of the system.

**Milord II** modular language is based on modules and generic modules. Generic modules allows us to save code and to make more understandable the code of an application. Generic modules can be instantiated dynamically.

A set of operations deals with incremental programming of applications. Refinement, contraction and expansion of modules allows the expert to build several versions of modules that are progressively refined and modified. **Milord II** considers all the versions are executable entities allowing an incremental validation and testing of the applications.

After the description of the modular language the following Chapters are devoted to the internal components of the modules giving an incremental description of the syntax and semantics of the complete **Milord II** system.

# Chapter 3

# Approximate Reasoning

We have seen in Chapter 2 that the natural form of describing and solving problems was by means of the decomposition of the problems into subproblems. As shown **Milord II** deals with structured problem solving integrating and adapting known modularization techniques into a reach ES shell.

A module of **Milord II** contains the necessary components to describe the domain and control knowledge relative to a problem, by means of facts, rules, metarules, and so on. Till now we have not described these internal components of a module. We only have considered that a module is able to produce items of information (export interface) from others items of information that has been provided by the user (import interface) or by the submodules of that module (hierarchy).

This Chapter is devoted to explain the nature of those items of information that we name *facts* in ES's terminology. For that we can consider that an item of information is composed of four components, that is, object, attribute, value and confidence (Dubois and Prade, 1988). The attribute is a function that attaches a value or a set of values to an object. The confidence indicates the reliability of an item of information.

Now we analyze the concepts of imprecision and uncertainty of an item of information. The imprecision is a concept attached to the value component of an item of information, and the uncertainty to the confidence one. For instance, we can say *It is very possible that the temperature of the patient is between* $36\,^{\circ}$ *and* $38^{\circ}$. In this example the object is the patient and the attribute is the temperature. The value of the temperature is imprecise and it belongs to the interval $[36^{\circ}, 38^{\circ}]$ with a confidence degree of *very possible*. This is an example of an imprecise and uncertain proposition. We can think in other examples, a precise and certain proposition: *The temperature of the patient is* $36.7\,^{\circ}$; an imprecise and certain proposition: *The temperature of the patient is between* $36\,^{\circ}$ *and* $38^{\circ}$; and finally a precise and uncertain proposition: *It is very possible that the temperature of the patient is* $36.7\,^{\circ}$.

The management of uncertainty and imprecision becomes essential to modelize real problems. Usually the kind of knowledge domains treated by ES's

contains imperfect knowledge. We can find many sources of imprecision and uncertainty. For instance, measurement devices have imprecise results (the corporal temperature depends on which part of the body we take the measure); subjective appreciations are imprecise (the valuation of the temperature touching the patient with the hand); the incomplete knowledge added to the natural language ambiguity produce uncertain propositions (*If the patient has headache it is possible that his temperature could be greater than* 37°).

In this Chapter we will explain how **Milord II** manages uncertainty and imprecision. As we will see it is limited to manage uncertain data or imprecise data. It is not possible to express at the same time the uncertainty and the imprecision of an item of information.

When we want to manage uncertainty by means of linguistic terms (for instance, *likely*, *possible*, *false*, and so on) we must decide how many terms to use and which they are. Usually these questions are problem dependent and they are related to the concrete meaning that the expert associates to these linguistic terms. Some problems could need five terms to express its uncertainty and another problem could need seven terms. Furthermore it is possible that the term *possible* would have a different meaning for different experts, or that one expert prefers to use another term such as *likely*. The concrete set of terms used to express uncertainty and its granularity will depend on the expert criteria and on the concrete problem he is considering.

As seen in Chapter 2 each module is used to specify a subproblem. It could be very useful to change the language of representation of uncertainty in function of the type of subproblem. The modular structure of **Milord II** together with its approach to uncertainty management, allows to define in a natural way local uncertainty calculus attached to each module, in such a way that the knowledge adequation process can also be applied to the uncertainty management. The interest in having different uncertainty calculus in an ES becomes clearer when expert systems involving several human experts have to be built. **Milord II** system allows us to define local uncertainty logics in the modules and mechanisms of communication among these logics(Agustí et al., 1992).

This Chapter is divided in four parts. The first one is devoted to the definition of algebras of truth–values. The expert must define which is the logic more adecuated to his problem. This implies to define a set of linguistic terms useful to express his knowledge (for instance, he would say *The confidence degree that the patient has pneumonia is* very possible or *The treatment with ciprofloxacine is* good). Furthermore the expert can express some control to the meaning of the logic operators. An example of this is to find the confidence degree of the fact *fever* from two sentences: *It is slightly possible that the patient has headache* ; *If the patient has headache it is possible that he has fever.*

The second part contains an explanation in depth of the treatment given by **Milord II** of the concepts of uncertainty and imprecision. This is reached by extending the algebra of truth–values to an algebra of intervals of truth–values (we can say that *The evidence that the patient has pneumonia is* between *possible and very possible*) and introducing fuzzy sets (for instance, *The degree*

*of membership of the patient to the set of the tall people is quite*).

The third part is devoted to the communication between the local logics of modules. When two modules use different logics (linguistic terms and operators) we must find a mechanism of translation of the linguistic terms to make compatible the communication between those modules (for instance, what is the meaning of the sentence *Fever is possible* for another module that do not use this term, maybe *Fever is likely*?). Finally we introduce all the constructs of the language **Milord II** that allow the experts to define the local logics of modules.

## 3.1 Algebra of truth–values

Psychological experiments (Kuipers et al., 1988; Fox, 1989) show that human problem solvers do not use numbers to deal with uncertainty and that the way they manage it is situation dependent. These requirements were partially satisfied in the *Milord* system in the sense that the treatment of uncertainty was based in different operators defined over a set of linguistic terms describing the global verbal scale the experts use to express degrees of uncertainty (Godo et al., 1989).

The approach used in **Milord II** to manage uncertainty is based on many–valued logics. The use of many–valued logics has been criticized (Pearl, 1990; Hàjek et al., 1992) because of the confusion between uncertainty and imprecision. We do not enter in depth in this kind of problems that has been extensively treated in the literature. Despite this, many–valued logics have been proved to be useful in ESs (Turner, 1984; Bonissone et al., 1987; Godo et al., 1989; López de Mántaras, 1990). We justify the use of many–valued logics because they have an efficient and simple deduction from the computability point of view. We will extend these logics with intervals of truth–values (Esteva et al., 1994) as a technique to manage uncertainty or imprecision. As we will see intervals of truth–values allow us to deal with uncertain or imprecise information and to introduce negative evidence in the sentences.

Here we present the definition of a family of many–valued logics which deductive system is based on a new kind of inference rule called specialization (it will be presented in the next Chapter). Some aspects of these logics have been already described in (Agustí et al., 1991; Agustí et al., 1992). Each logic is determined by a particular algebra of truth–values from a parametric family that is described next.

**Definition 3.1 (Algebra of Truth–values)** *An algebra of truth–values is a finite algebra*

$$\mathcal{A}_T^n = < A_n, N_n, T, I_T >$$

*such that:*

1. *The ordered set of truth–values $A_n$ is a chain of $n$ elements:*

$$0 = a_1 < a_2 < \cdots < a_n = 1$$

   *where $0$ and $1$ are the boolean False and True respectively.*

2. *The negation operator* $N_n$ *is the unary operation defined as*

$$N_n(a_i) = a_{n-i+1}$$

*the only one that fulfills the following properties:*

**N1:** *If* $a < b$ *then* $N_n(a) > N_n(b)$, $\forall a, b \in A_n$

**N2:** $N_n^2 = Id$.

3. *The conjunction operation* $T$ *is a 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.*

$$I_T(a, b) = \; Max \; \{c \in A_n | T(a, c) \leq b\}$$

*and 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* $I_T(c, a) \leq I_T(c, b)$ *for all* $c$

**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–values algebras is completely determined as soon as the set of truth–values $A_n$ and the conjunction operator $T$ are determined. So, varying these two characteristics we can obtain a parametric family of different many–valued logics, including, among others, Kleene's and Łukasiewicz's logics.

As we can see in the example of the Figure 3.1 (the complete definition for the module *Gram_of_Sputum* given in the Figure 2.8 and used in the previous examples), this module declares a local logic (inference system) consisting in a set of linguistic terms (truth values) and the conjunction operator (conjunction).

The definition of this local logic in the module *Gram_of_Sputum* is determined by the declaration of the set of five linguistic term $S_5 = \{$false, unlikely, may_be, likely, true$\}$, and the conjunction operation $T_{S_5}$ operation defined in the Table 3.1. The expert has chosen this number of terms because he considers that it is sufficient to talk about the concepts contained in this module. The names

```
Module Gram_of_Sputum =
    Begin
    Import Sputum_clas, Sputum_Gram
    Export End, Gram_yes, DCGP, CGPC, CGPR, BGN, CBGN
    Deductive knowledge
    Dictionary: not defined here
        Rules :
        R001 If Sputum_clas=(Grup_1 or Grup_2 or Grup_3)
                then conclude Sputum_ok is true
        R002 If Sputum_clas=(Grup_4 or Grup_5 or Grup_6)
                then conclude Sputum_not_ok is true
        R003 If Sputum_ok then conclude Gram_yes is true
        R004 If Sputum_not_ok then conclude End is true
        R005 If Sputum_Gram=(DCGP_MC) then conclude DCGP is true
        R006 If Sputum_Gram=(DCGP_MC) then conclude CGPC is unlikely
        R007 If Sputum_Gram=(CGPC_MC) then conclude CGPC is likely
        R008 If Sputum_Gram=(CGPC_MC) then conclude DCGP is unlikely
        R009 If Sputum_Gram=(CGPR_MC) then conclude CGPR is true
        R010 If Sputum_Gram=(BGN_MC) then conclude BGN is likely
        R011 If Sputum_Gram=(CBGN_MC) then conclude CBGN is likely
        Inference system:
        Truth values = (false, unlikely, may_be, likely, true)
        Connectives :
            Conjunction = Truth Table
                ((false, false, false, false,false)
                (false, unlikely, unlikely,unlikely, unlikely)
                (false, unlikely, may_be, may_be, may_be)
                (false, unlikely, may_be, may_be, likely)
                (false, unlikely, may_be, likely, true)))
    End deductive
    End
```

Figure 3.1: Example of Local logic declaration.

| | false | unlikely | may_be | likely | true |
|---|---|---|---|---|---|
| false | false | false | false | false | false |
| unlikely | false | unlikely | unlikely | unlikely | unlikely |
| may_be | false | unlikely | may_be | may_be | may_be |
| likely | false | unlikely | may_be | likely | likely |
| true | false | unlikely | may_be | likely | true |

Table 3.1: $T_{S_5}$ Table.

of the linguistic terms used in each module are intended to make understandable the kind of the concepts that are used into the modules.

The expert can choose a $T$ function (holding the properties **T1**–**T5**) depending on the meaning he wants to give to the conjunction operation[1]. For instance, considering that $T(a, b) \leq min(a, b)$ (from properties **T4** and **T5**) the expert can consider more or less optimistic evidence combinations. For instance, he can consider that the conjunction of two propositions with confidence degree *likely* results on a confidence degree of *may_be* (less than *likely*).

| | $N_n$ |
|---|---|
| false | true |
| unlikely | likely |
| may_be | may_be |
| likely | unlikely |
| true | false |

Table 3.2: $N_5$ Table.

It is easy to see that the negation operator $N_5$ and the implication operator $I_{T_{S_5}}$ can be deduced from the above definition of the algebra of truth–values (see the Tables 3.2 and 3.3 respectively).

| x / y | false | unlikely | may_be | likely | true |
|---|---|---|---|---|---|
| false | true | true | true | true | true |
| unlikely | false | true | true | true | true |
| may_be | false | unlikely | true | true | true |
| likely | false | unlikely | likely | true | true |
| true | false | unlikely | may_be | likely | true |

Table 3.3: $I_{T_{S_5}}(x, y)$ Table.

### 3.1.1 Modus Ponens Operator

The language used in **Milord II** is composed of weighted facts and rules. Till now we have only introduced the conjunction and negation operators. Then we should also introduce the many–valued version of modus ponens inference rule (Alsina et al., 1984; Valverde and Trillas, 1985; Trillas and Valverde, 1987) that allows us to make deductions, that is, to deduce the truth–value of the conclusion of a rule from the truth–values of its conditions and the truth–value of the rule.

[1]A general algorithm for finding truth–value algebras over a partially ordered set of $n$ elements is given in (Godo and Meseguer, 1991).

**Definition 3.2 (Modus Ponens)** *$MP_T$ is a function from $A_n \times A_n$ to the set of intervals*[2] *of $A_n$ defined as:*

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

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

This is a functional expression of the multiple–valued version of the classical modus ponens rule, i.e. $MP_T(a,b)$ is the set of solutions for $\rho(q)$ in the equation system:

$$\begin{cases} \rho(p) = a \\ \rho(p \to q) = I_T(\rho(p), \rho(q)) = b \end{cases}$$

where $\rho$ is the valuation of the sentences.

The modus ponens table for $S_5$ and $T_{S_5}$ is given in Figure 3.4. For instance, if the value of the fact $p$ is *likely* and the value of the rule $p \to q$ is *true*, by means of the definition of the $MP_T$ function and the Table $T_{S_5}$ we can deduce that the fact $q$ has the value *likely*. We will return over this rule when we explain the specialization of KB's in Chapter 4.

| x / y | false | unlikely | may_be | likely | true |
|---|---|---|---|---|---|
| false | ∅ | ∅ | ∅ | ∅ | [false,true] |
| unlikely | false | ∅ | ∅ | ∅ | [unlikely,true] |
| may_be | false | unlikely | ∅ | ∅ | [may_be,true] |
| likely | false | unlikely | ∅ | may_be | [likely,true] |
| true | false | unlikely | may_be | likely | [true, true] |

Table 3.4: $MP_{T_{S_5}}(x,y)$ Table.

We know how to use a propositional language (no defined yet) with negation, conjunction and implication operators. Till now we only have talked about the deduction by means of the modus ponens inference rule. In Chapter 4 we will introduce a different kind of deduction based on the specialization inference rule (SIR).

## 3.2 Uncertainty and Imprecision

We have considered in the introduction that uncertainty and imprecision take an important role in the data used by ESs. Now we will talk on the concrete

[2]Notice that the modus ponens operator can return an interval of truth–values. This will be one of the reasons to introduce intervals of truth–values as we will see.

items of information that can be used in **Milord II**. Our system deals with uncertainty and imprecision by means of the use of intervals of linguistics terms.

Consider the following two examples:

1. *Peter is quite tall.*
2. *It is possible that Peter is tall.*

These two propositions can be confusing. When we read the first proposition we could think that the fact of being *tall* is a graduated measure, then we are talking about the value of the attribute tall. There are persons that are clearly tall (for instance, those that their height is greater than 2m.) and persons that are clearly not tall, but between these two categories there are persons that are slightly tall, quite tall, etc.

Another interpretation can guide us to the concept of fuzzy sets. The difference between a classic set (named crisp set) and a fuzzy set is that the elements of the considered universe have a degree of membership to the fuzzy set. In crisp sets the elements of the universe belong or not belong to the crisp set. We can consider that the concept *tall* is a fuzzy set. We can imagine then a characteristic function that given the height of a person it returns the degree of membership of the person to the fuzzy set tall (see the Figure 3.2), expressed by means of linguistic terms (like *quite*).

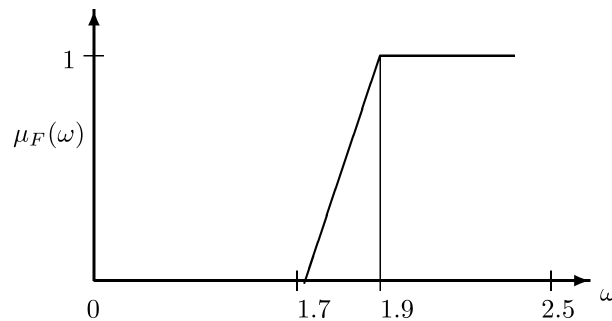


Figure 3.2: Fuzzy set representing the concept *tall*.

The second proposition can be interpreted as the first one, but its syntactical form guides us to think that tall is a boolean concept (a person is tall or not tall) and we have a degree of confidence on the fact that Peter is tall. In this case we are talking about uncertainty. We are not sure if Peter belongs or not to the set tall.

There are other examples where that confusion is not possible, for instance, we can say that *It is possible that Peter has pneumonia* but we can not talk

about degrees of pneumonia (it is not possible to say that somebody is quite pneumonic). These interpretations depend on the concept and the concrete linguistic terms we are using.

In **Milord II** we will consider only information of the second type, that is, uncertain information expressed by means of linguistic terms, but the semantical interpretation of those sentences will be dependent on the set of concepts the expert is using. We will take advantage of this confusion[3] to introduce imprecision.

### Imprecision at truth level

**Milord II** introduces imprecision at the truth level by means of intervals of linguistic terms. One may know that a proposition must take a truth–value but does not know exactly which, then a set of possible values is given. We can say that *The confidence degree of pneumonia is between possible and very_possible.*

This kind of imprecision could seem not to be very useful. We can think that users would have difficulty to interpret these intervals of truth–values, but we will make it clear by comparing the interpretation of these intervals in **Milord II** with the interpretation of truth–values in *Milord*.

In *Milord* all the truth–values associated to the facts were considered to be positive evidences. When we found more than one deductive path to evaluate a fact, we choose the best, that is, that of the greatest truth–value. In *Milord* the goal was to obtain the maximum certainty degree for the facts. Furthermore in the case that the fact would get a truth–value less than a threshold, it would be considered unknown (there was no sufficient positive evidence to consider it).

Now we can comment the interpretation of positive evidence by means of intervals of truth–values. We consider that positive evidence is an interval from a truth–value to true, that is, when we say that the value of the fact *pneumonia* is *possible*, the interpretation is that the value of *pneumonia* belongs to the interval $[possible, true]$. When a fact belongs to the interval $[0, 1]$ it means that we know nothing about its value. With this interpretation when we obtain the value *false* for a fact it means that the value of this fact is *unknown* instead of false.

Thanks to this interpretation we can introduce negative evidence as intervals of truth–values of the form $[0, a_i]$, that is, the value of the fact could be from false to $a_i$. In **Milord II** we interpret positive and negative evidence as before. We can find deductive paths with positive and negative evidences. The result of the parallel combination is the fusion (intersection) of these intervals (as we will see in Section 4.2.3). Then the interpretation of an interval as $[a_i, a_j]$ is that the positive evidence is $a_i$, but there are a negative evidence $a_j$. The goal of **Milord II** is to obtain the maximum precision of the value of facts. We will return on this in the following Sections.

[3]Experience shows that experts use this kind of interpretation. Depending on the set of concepts they are using the interpretation of the linguistic terms could be taken as values or confidence degrees.

**Imprecision at value level**

If we return to the examples above we can interpret the imprecision at truth level as imprecision at value level, the more standard one. When we say that *It is between possible and very possible that Peter is tall* we could think of a previous imprecise measure of the height of Peter (a numeric interval). The characteristic function of the fuzzy set *tall* is a function from intervals of height to intervals of degrees of membership to the fuzzy set tall.

In the following Section we will present the extension of the algebra of truth–values to the algebra of intervals. Now we consider that the sense of these intervals is the imprecision at truth level. We will return on imprecision at value level in Section 4.4.1.

### 3.2.1 Intervals of Truth–values

We have presented several semantic motivations to introduce intervals of truth–values, but there are other reasons. The first one is related to the chaining of rules. We can see in the above definition of the modus ponens inference rule that in some cases it returns an interval of truth–values. The second reason is to make possible the mappings between different local logics and it will be explained in Section 3.3.

The deductive system of **Milord II** works on the set of intervals of $A_n$, $Int(A_n)$. $N_n^*$, $T^*$ and $MP_T^*$ are the point–wise extensions[4] of $N_n$, $T$ and $MP_T$ respectively. Let us introduce more formally the intervals of truth–values and these operators.

**Definition 3.3 (Intervals)** *$Int(A_n)$ will stand for the set of intervals of $A_n$, i.e.*

$$Int(A_n) = \{[a, b] | a, b \in A_n \text{ such that } a \leq b\} \cup \emptyset$$

*being $[a, b] = \{c \in A_n | a \leq c \leq b\}$.*

Notice that there exists an embedding from $A_n$ to $Int(A_n)$ identifying elements $a$ with intervals $[a, a]$ of $Int(A_n)$. For the sake of simplicity on the notation, we will also denote by $a$ the interval $[a, a]$ when no confusion is possible. The interval $\emptyset$ represents an inconsistent value as we will see.

On the set of intervals $Int(A_n)$ we can define the imprecision ordering and the uncertainty ordering (Esteva et al., 1994).

**Imprecision ordering:** Is naturally induced by the set inclusion relationship $\subseteq$. Given $A, B \in Int(A_n)$ , the interval $A$ is more precise or equal than $B$, if and only if $A \subseteq B$ (see Figure 3.3 for an example on $Int(A_4)$, where $A_4 = \{0, a, b, 1\}$). The goal of **Milord II** is to produce the most precise values.

**Uncertainty ordering:** The uncertainty order of $A_n$ can be extended into $Int(A_n)$ in at least two different ways:

[4]Strictly speaking they give the minimal interval containing the point–wise extensions.

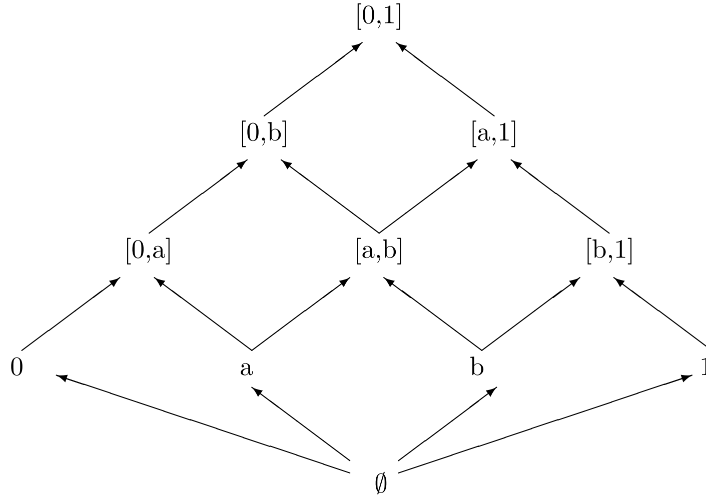


Figure 3.3: Imprecision Ordering on $Int(A_4)$.

- Weak uncertainty order (see Figure 3.4):

$$[a_1, b_1] \tilde{\leq} [a_2, b_2] \text{ if, and only if, } a_1 \leq a_2 \text{ and } b_1 \leq b_2$$

- Strong uncertainty order:

$$[a_1, b_1] \preceq [a_2, b_2] \text{ if, and only if, } b_1 \leq a_2 \text{ or } [a_1, b_1] = [a_2, b_2]$$

Now we will define the negation, conjunction and modus ponens operators for the intervals of truth–values.

**Definition 3.4** *We define on the set of intervals of $A_n$ the functions $N_n^*$ and $T^*$ as those functions that give the minimal interval containing the point–wise extensions of $N_n$ and $T$ respectively. That is:*

- $N_n^*([a, b]) = [N_n(b), N_n(a)]$
- $T^*([a, b], [c, d]) = [T(a, c), T(b, d)]$

In order to get a functional expression of the multiple–valued version of the Modus Ponens rule, we also define on the set of intervals of $A_n$ the function $MP_T^*$ as follows:

**Definition 3.5** *For any truth–intervals $V$ and $W$, we define $MP_T^*(V, W)$ as the minimal interval containing all solutions for $z$ in the family of functional equations*

$$I_T(a, z) = b$$

*varying $a \in V$ and $b \in W$.*

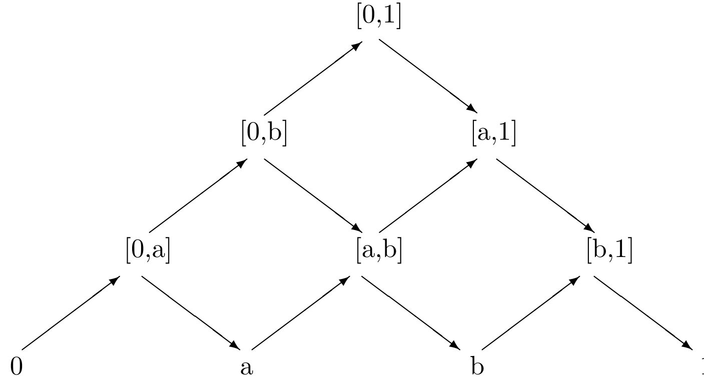


Figure 3.4: Weak Uncertainty Ordering on $Int(A_4)$.

This definition can be made more explicit when taking into account that the truth–intervals $W$ attached to rules of **Milord II** are always upper intervals, i.e. $W$ is of the form $W = [c, 1]$.

**Proposition 3.1** $MP_T^*([a, b], [c, 1]) = [T(a, c), 1]$

After the introduction of the intervals and its operators we will explain briefly how we work with intervals and the concept of precision.

### 3.2.2 Working with intervals

We have seen the extension to an algebra of intervals of truth–values. Now we can explain how these intervals are used in practice with the rules of **Milord II**.

The goal of *Milord* system was to find the maximum truth–value for the facts because it only uses rules with positive evidence. Because of the use of intervals of truth–values, the goal of **Milord II** is to obtain the maximum precision interval for the facts (combining positive and negative evidences).

Consider a set of rules with the same fact in the conclusion, but some rules with the negation of that fact in their conclusion. Then we should consider the following points:

- The rules with a positive conclusion (not negated) will fix the minimum certainty value of the conclusion fact.
- The rules with a negative conclusion will fix the maximum certainty value of the conclusion fact.
- We combine evidences by means of the intersection of intervals.

We can explain it through an example. Consider that the weighted sentences are pairs $(sentence, value)$. And consider the following sentences (two facts and two rules)[5]:

$$\left.\begin{array}{l}(a, \rho_a)\\(b, \rho_b)\\(a \rightarrow c, [\rho_{r_1}, 1])\\(b \rightarrow \neg c, [\rho_{r_2}, 1])\end{array}\right\}$$

From these facts and rules we can obtain a value for the sentence $c$ and for $\neg c$ using the modus ponens inference rules for intervals of truth–values. Notice that the value of a **Milord II** rule is an upper–interval to 1. The values for the sentences in the conclusions of those rules are then upper–intervals.

$$\left.\begin{array}{l}(a, \rho_a)\\(b, \rho_b)\\(a \rightarrow c, [\rho_{r_1}, 1])\\(b \rightarrow \neg c, [\rho_{r_2}, 1])\\(c, MP^*_T([\rho_a, \rho_a], [\rho_{r_1}, 1])) = (c, [T(\rho_a, \rho_{r_1}), 1])\\(\neg c, MP^*_T([\rho_b, \rho_a], [\rho_{r_2}, 1])) = (\neg c, [T(\rho_b, \rho_{r_2}), 1])\end{array}\right\}$$

Using the negation operator of intervals of truth–values we obtain an interval from 0 for the conclusion of the second rule. As commented before the positive rule produces a positive evidence of the conclusion and the negative rule a negative one. The fusion (intersection) of intervals results in a interval where its minimum value corresponds to a positive evidence and the maximum value to the negative one.

$$\left.\begin{array}{l}(c, [T(\rho_a, \rho_{r_1}), 1])\\(\neg c, [T(\rho_b, \rho_{r_2}), 1]) \Rightarrow (c, [0, N_n(T(\rho_b, \rho_{r_2}))])\end{array}\right\} \Rightarrow$$

$$(c, [T(\rho_a, \rho_{r_1}), N_n(T(\rho_b, \rho_{r_2}))])$$

When the intersection of values is empty, then it is considered to be inconsistent. All the operators $N^*_n$, $T^*$ and $MP^*_T$ with an argument equal to $\emptyset$ return the same inconsistent value $\emptyset$.

### 3.2.3 Fuzzy Sets

Facts are one of the most primitive component of **Milord II** that represent the concepts we will use in an ES. To deal with uncertainty and imprecision we associate an interval of truth–values to the facts. Sometimes it is interesting to deal with other type of facts that represents set concepts. For instance, we can consider that the fact *treatment* is a set of *antibiotics*. We could be interested in comparing different *treatments* by means of set relations and operations.

From the logical point of view the concept of fuzzy set is carried out by changing the usual definition of the characteristic function of a set by means of

[5]Notice that *Milord* did not use negation in the conclusion of rules.

degrees of membership. We use uncertainty in the knowledge we possess about the membership relation. To define a fuzzy set $F$ we give a reference set $\Omega$ and a mapping, $\mu_F(\omega)$, of $\Omega$ into $[0,1]$. $\mu_F(\omega)$, for $\omega \in \Omega$ is interpreted as the degree of membership of $\omega$ in the fuzzy set $F$. When $\mu_F(\omega) \in \{0,1\}, \forall\omega$, $F$ is the same as an ordinary subset of $\Omega$. This is called a *crisp subset* of $\Omega$ and is a particular case of fuzzy sets.

That is the usual definition of fuzzy sets as a mapping of the characteristic function into the interval $[0,1]$. Because of the use of linguistic terms and imprecision we will extent the usual definition to the intervals of truth–values.

Now we put an example of fuzzy set named *treatment*. Consider the following reference set, in this case a set of antibiotics:

$\Omega =$ {carbamacepina, teofilina, digoxina ,dicumarinics, ciclosporina, difenilhidantoina}

We use as imprecise degree of membership intervals of linguistic labels $Int(A_n)$ of the logic of the module that contains this fact. We can specify a fuzzy set by means of its characteristic function. For instance, we consider that a *treatment* is a fuzzy set composed by antibiotics[6]. The characteristic function of the fact *treatment* could be:

$\mu_{treatment}$(carbamacepina) = [impossible, impossible]
$\mu_{treatment}$(teofilina) = [definite, definite]
$\mu_{treatment}$(digoxina) = [impossible, definite]
$\mu_{treatment}$(dicumarinics) = [very_possible, very_possible]
$\mu_{treatment}$(ciclosporina) = [possible, very_possible]
$\mu_{treatment}$(difenilhidantoina) = [impossible, impossible]

After the definition of this kind of facts representing fuzzy sets we should say how to use them in the rules. We can use them by comparing different fuzzy sets. For instance, consider two different *treatments* in the universe of *antibiotics*. We can say *If treatment1 is a subset of the treatment2 then ...* or *If the treatment1 intersects with the treatment2 then ....* This kind of operations and relations results in an interval of truth–values as we will see.

### Elementary Operations and Relations

Here we explain the possible operations and relations among fuzzy sets used in **Milord II**. They are based on usual concepts used in fuzzy set theory and operations among fuzzy sets (Zadeh, 1965). This provides to the experts a complete tool to work with fuzzy sets. First of all we describe the unary operations using the above example.

**Cut:** We can use a threshold $\alpha$ to obtain a set $F_\alpha$ called $\alpha$*–cut* of $F$. The definition is the following:

$$F_\alpha = \{\omega \in \Omega | min(\mu_F(\omega)) \geq \alpha\}$$

[6]This kind of facts like *treatment* are named enumerated facts in **Milord II**.

$F_\alpha$ contains all the elements of $\Omega$ that are compatible with $F$ at level at least $\alpha$. In **Milord II** $\alpha$ is the threshold of the module (see the Section 5.2) that contains the fact $F$. The threshold in the modules express the minimum certainty value that is considered to be significant. Suppose $\alpha$ to be *possible* in the above example, then it is easy to see that:

$treatment_{possible}$ = {teofilina, dicumarinics, ciclosporina}

**Core:** The cut of $F$ at level 1 is called the *core* of $F$, denoted $\dot{F}$.

$$\dot{F} = \{\omega \in \Omega | \mu_F(\omega) = 1\}$$

It contains all the elements of $\Omega$ that are in $F$ at level 1. Following the same example we can see that:

$\dot{treatment}$ = {teofilina}

**Support:** The support set contains all the elements of $\Omega$ that are not at level 0.

$$S(F) = \{\omega \in \Omega | min(\mu_F(\omega)) > 0\}$$

In our example they are all the elements with certainty value greater than *impossible*.

$S(treatment)$ = {teofilina, dicumarinics, ciclosporina}

**Complement:** Complementation of set $F$ is defined as the set $\bar{F}$ that has the following characteristic function:

$$\forall\omega, \mu_{\bar{F}}(\omega) = N_n^*(\mu_F(\omega))$$

In the same example the characteristic function associated to the set $\bar{treatment}$ is:

$$\mu_{\bar{treatment}}(carbamacepina) = [definite, definite]$$
$$\mu_{\bar{treatment}}(teofilina) = [impossible, impossible]$$
$$\mu_{\bar{treatment}}(digoxina) = [impossible, definite]$$
$$\mu_{\bar{treatment}}(dicumarinics) = [slightly_possible, slightly_possible]$$
$$\mu_{\bar{treatment}}(ciclosporina) = [slightly_possible, possible]$$
$$\mu_{\bar{treatment}}(difenilhidantoina) = [definite, definite]$$

We consider the normal union and intersection operations among fuzzy sets:

**Union:** $\forall\omega, \mu_{F\cup G}(\omega) = max^*(\mu_F(\omega), \mu_G(\omega))$

**Intersection:** $\forall\omega, \mu_{F\cap G}(\omega) = min^*(\mu_F(\omega), \mu_G(\omega))$

These operations are compatible with those on *crisp* sets[7]. Finally we explain the set of relations we can use in **Milord II**. These relations given two fuzzy sets return an interval of truth–values.

$$R : \tilde{\mathcal{P}}(U) \times \tilde{\mathcal{P}}(U) \rightarrow Int(A_n)$$

Before that we define the combination of relations and the complement of a relation as:

**Combination:** $(R * S)(F, G) = min^*(R(F, G), S(F, G))$

**Complement:** $\bar{R}(F, G) = N_n^*(R(F, G))$

Now we define the inclusion, intersection and equality between two fuzzy sets and its meaning.

**Inclusion or equal:**
$R_{\subseteq}(F, G) = min^*(\mu_{\bar{F} \cup G})$
$R_{\supseteq}(F, G) = R_{\subseteq}(G, F)$

**Intersection degree:**
$R_{\cap}(F, G) = max^*(\mu_{F \cap G})$

**Equality:** $R_{=}(F, G) = (R_{\subseteq} * R_{\supseteq})(F, G)$

**Inclusion:**
$R_{\subset}(F, G) = (R_{\subseteq} * \bar{R}_{=})(F, G)$
$R_{\supset}(F, G) = R_{\subset}(G, F)$

These relations return degrees of inclusion, intersection and equality between two fuzzy sets. Using them we can express the degree of intersection of two treatments, as so on.

All these operations are the most standard (Zadeh, 1965) and they are compatible with crisp sets. In these definitions we have used for simplicity the functions *min* and *max*, but we can use instead the more general ones, the triangular norm $T$ and the triangular conorm $S(x, y) = N(T(N(x), N(y)))$.

## 3.3 Local Logics

The need of communication among modules with different local uncertainty calculus has lead us to analyze the correspondence between uncertainty calculus. Uncertainty calculus can be considered as inference mechanisms defining logical entailment relationships. Therefore the correspondences (or communication) between different calculus can be analyzed as mappings between different entailment systems. To do that we will introduce a summary of the main theoretical results on mappings between entailment systems (Agustí et al., 1992) and we propose an algorithm to find the mappings that allow modules to communicate.

[7]We consider the functions maximum and minimum as the point–wise extensions: $max^*([a, b], [c, d]) = [max(a, c), max(b, d)]$ and $min^*([a, b], [c, d]) = [min(a, c), min(b, d)]$

### 3.3.1 Mappings between different local logics

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

**A)** If $\Gamma \vdash e$, then $H(\Gamma) \vdash' H(e)$

With this requirement we assure that for every formula deducible from a set of formulas $\Gamma$ in $M$, its corresponding formula in $M'$ by the mapping, $H(e)$, will also be deducible in $M'$ from the corresponding formulas of $H(\Gamma)$. In other words, there is no inferential power lost when translating from $M$ to $M'$ through a mapping $H$ satisfying **A**. Nevertheless the main drawback of requirement **A** is that it does not forbid to deduce from $H(\Gamma)$, in $M'$, formulas that are not translations of any formula deducible from $\Gamma$ in $M$. The property means that, in the case of modules representing different experts, an expert $E'$ related to $M'$, using knowledge coming from an expert $E$ related to $M$, will be able to deduce the same facts than $E$, but not only those facts. We need:

**B)** If $H(\Gamma) \vdash' H(e)$, then $\Gamma \vdash e$

This is the inverse requirement of **A**. So, in this case all deductions in $M'$ involving only translated formulas from $M$ are translations of deductions in $M$, or equivalently if a fact is not deducible in $M$, then its correspondent fact in $M'$ will neither be deducible from the translated knowledge. An alternative to **B** could be:

**C)** If $H(\Gamma) \vdash' e'$, then there exists $e$ such that $\Gamma \vdash e$ and $H(e) \vdash' e'$.

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

It is worth noticing that, if $C$ denotes the consequence operator with respect to an entailment system $(L, \vdash)$, that is, $C(\Gamma) = \{e \in L | \Gamma \vdash e\}$ for all set of formulas $\Gamma$, then the requirements **A** and **B** can be rewritten in the following way:

**A)** $H(C(\Gamma)) \subset C'(H(\Gamma))$

**B)** $C'(H(\Gamma)) \subset H(C(\Gamma))$

being $C'$ the consequence operator associated to the entailment system $(L', \vdash')$.

From these three different requirements we can define the conditions on the mappings in order these requirements hold. Notice that they are mappings of truth–values algebras[8].

**Theorem 3.1** *Given two truth–values algebras* $\mathcal{A}^n_{T_1} =< A_n, N_n, T_1, I_{T_1} >$ *and* $\mathcal{B}^m_{T_2} =< B_m, N_m, T_2, I_{T_2} >$*, a mapping* $H : A_n \to I(B_m)$ *fulfills the requirements* **A** *or* **B** *or* **C** *from* $\mathcal{A}^n_{T_1}$ *to* $\mathcal{B}^m_{T_2}$ *if the following conditions hold:*

1. $H$ *is non–decreasing, i.e. if* $a \leq b$*, then* $H(a) \leq^* H(b)$
2. $H(0) = 0$
3. $H(N_n(x)) = N^*_m(H(x))$
4. *We have one case for each requirement:*

   **A)** $H(T_1(x,y)) \supset T^*_2(H(x), H(y))$

   **B)** $H(T_1(x,y)) \subset T^*_2(H(x), H(y))$
   $H(x) \subset H(y) \Rightarrow x = y$

   **C)** $H(T_1(x,y)) \subset T^*_2(H(x), H(y))$

Finally we explain the algorithm that allows us to find mappings that are quasi–morfisms.

We can divide every chain $A_n$ in three subsets:

- $\mathcal{N}_n = \{x | x < N_n(x)\}$
- $\mathcal{F}_n = \{x | x = N_n(x)\}$
- $\mathcal{P}_n = \{x | x > N_n(x)\}$

We consider three subsets $\mathcal{N}^*_n$, $\mathcal{F}^*_n$ and $\mathcal{P}^*_n$ for the case of intervals $Int(A_n)$. In that case we use the weak uncertainty order defined before.

- $\mathcal{N}^*_n = \{x | x \tilde{<} N_n(x)\}$
- $\mathcal{F}^*_n = \{x | x = N_n(x)\}$
- $\mathcal{P}^*_n = \{x | x \tilde{>} N_n(x)\}$

After that we find all the mappings

$$H_i : \mathcal{N}_n \cup \mathcal{F}_n \to \mathcal{N}^*_m \cup \mathcal{F}^*_m$$

such that:

1. $H_i(0) = 0$

[8] The mappings are from the elements of a chain to elements of the set of intervals of the other chain.

2. $H_i(\mathcal{F}_n) \in \mathcal{F}^*_m$

3. If $x \leq y$ then $H_i(x) \leq^* H_i(y)$, where $x \in A_n$ and $y \in Int(B_m)$.

Now we can extend these mappings:

$$H(x) = \begin{cases} H_i(x) & \text{if } x \in \mathcal{N}_n \cup \mathcal{F}_n \\ N^*_m(H_i(H(x))) & \text{if } x \in \mathcal{P}_n \end{cases}$$

Finally we must check which mapping is quasi–morfisms for the three criteria, that is, checking the condition 4 of the quasi–morfism definition.

### 3.3.2 Example

For this example we will consider the set of modules used as examples in Chapter 2. Consider the module *Gram* (Figure 2.1) that has four submodules *Respiratory_Diagnosis*, *Type_of_Infection*, *Previous_Treatment* (see Figure 2.3) and *Gram_of_Sputum* (Figure 2.8).

Consider the following sets of truth–values corresponding to these modules, $Gram_7$ for the module *Gram* and $D_2$, $T_2$, $P_2$ and $S_5$ for its submodules:

$Gram_7$ = {impossible, few_possible, sligh_possible, possible, quite_possible, very_possible, sure}
$D_2$ = {false, true}
$T_2$ = {false, true}
$P_2$ = {impossible, sure}
$S_5$ = {false, unlikely, may_be, likely, true}

and the $T$ functions $T_{S_5}$ and $T_{Gram_7}$ (see Tables 3.1 and 3.5 respectively). We do not represent the other $T$ functions because they are boolean logics with different names for *true* and *false*. This is the situation of Figure 3.5 where we need to find the mappings $H_T$, $H_S$, $H_P$ and $H_D$.

| | impos | few_p | sli_p | possib | quite_p | very_p | sure |
|---|---|---|---|---|---|---|---|
| impos | impos | impos | impos | impos | impos | impos | impos |
| few_p | impos | few_p | few_p | few_p | few_p | few_p | few_p |
| sli_p | impos | few_p | sli_p | sli_p | sli_p | sli_p | sli_p |
| possib | impos | few_p | sli_p | possib | possib | possib | possib |
| quite_p | impos | few_p | sli_p | possib | very_p | very_p | very_p |
| very_p | impos | few_p | sli_p | possib | quite_p | very_p | very_p |
| sure | impos | few_p | sli_p | possib | quite_p | very_p | sure |

Table 3.5: $T_{Gram_7}$ Table.

First we focuses over the more difficult mapping $H_S$. Following the above definitions we can see that the sets $\mathcal{N}_5$, $\mathcal{F}_5$, $\mathcal{N}^*_7$ and $\mathcal{F}^*_7$ are:

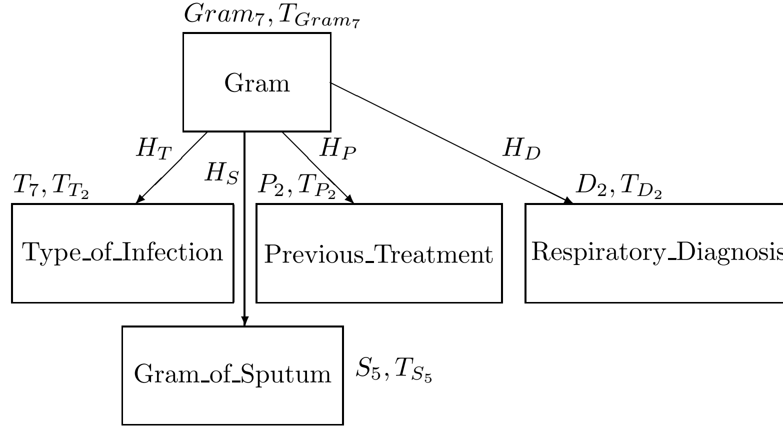


Figure 3.5: Mapping example.

$\mathcal{N}_5 = \{false, unlikely\}$
$\mathcal{F}_5 = \{may_be\}$
$\mathcal{N}_7^* = \{impos, few_p, sli_p, possib, [impos, few_p], [impos, sli_p], [impos, possib], [few_p, sli_p], [few_p, possib], [sli_p, possib]\}$
$\mathcal{F}_7^* = \{possib, [few_p, very_p], [sli_p, quite_p], [impos, sure]\}$

Here there are two mappings that are examples of those that hold the last requirement **C**:

$$\begin{cases} S/false \rightarrow impossible \\ S/unlikely \rightarrow [impossible, few_possible] \\ S/may_be \rightarrow [few_possible, very_possible] \\ S/likely \rightarrow [very_possible, sure] \\ S/true \rightarrow sure \end{cases}$$

$$\begin{cases} S/false \rightarrow impossible \\ S/unlikely \rightarrow [impossible, slightly_possible] \\ S/may_be \rightarrow possible \\ S/likely \rightarrow [quite_possible, sure] \\ S/true \rightarrow sure \end{cases}$$

The other cases are very simple because the other logics are boolean. Then we can map the first term of those logics to the first term of the logic of the module *Gram*. We can express the mapping as:

$$\begin{cases} D/false \rightarrow impossible \\ D/true \rightarrow sure \end{cases}$$

$$\begin{cases} P/impossible \rightarrow impossible \\ P/sure \rightarrow sure \end{cases}$$

Given a value from a submodule of the module *Gram* we can translate that value by means of the mappings above.

## 3.4 Logic Declaration

After the presentation of the local logics used in **Milord II**, here we describe the syntactical declaration of the inference system of a module. Each module contains the inference system declaration[9]. It contains the declaration of the concrete logic that will be used in a module and the mechanism of communication with the local logics of its submodules.

The inference system declaration contains the set of ordered truth–values $A_n$, the renaming mappings among the local logic and those of the submodules and a set of logic operators: negation, conjunction, disjunction and modus ponens (see Figure 3.6).

```
Inference system:
                Truth values = (...)
                Renaming:  ...
                Connectives:
                              Negation = ...
                              Conjunction = ...
                              Disjunction = ...
                Inference patterns:  ...
                              Modus ponens = ...
```

Figure 3.6: Logic declaration.

Now we will explain in depth these declarations specially to make clear the multiple possibilities of the language. Normally the logics used in **Milord II** are of the type explained till now, that is, the expert declare a set of truth–values and a $T$ function. To experiment with another kind of logics we allow to declare complete logics.

### 3.4.1 Truth values

Truth values of a module can be defined given an ordered set of symbols representing linguistic terms, where each symbol is a truth–value. For instance the declaration of the set $A_5$ is:

[9]As explained in Chapter 2 a module can declare its inference system by means of the inheritance mechanism.

**Truth values** = (impossible, sli_possible, possible, very_possible, sure)

Notice that given a truth value declaration, the first symbol is considered to have the semantics of *true* and the last symbol of *false*.

To maintain the logic declaration style that was used in *Milord* there are another possible form of declaring the truth–values. It consists in associating to each linguistic term four real numbers in the interval $[0, 1]$. Then we can use a truth–values declaration of the following form:

**Truth values** = (impossible = (a,b,c,d) , ... )

They correspond to the representation of a fuzzy interval (see the Figure 3.7 ) by means of a trapezoidal approximation of a fuzzy interval, as was used in *Milord*.

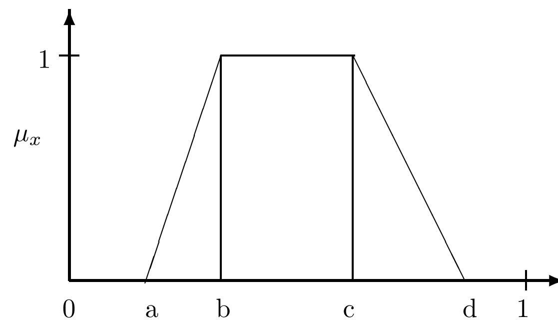


Figure 3.7: Trapezoidal approximation of a fuzzy interval.

### 3.4.2 Connectives

After the declaration of the set of truth–values of the local logic we must define the connectives and inference pattern of the logic, that is, the negation, conjunction, disjunction and modus ponens operators. To do that we have three options:

1. The standard one in **Milord II** is to define a set of truth–values and a conjunction connective by means of a table. Then the system generates the other connectives, that is, negation, disjunction and modus ponens as seen above.

2. We can declare the set of truth–values and all the connectives and modus ponens by means of tables. In this case these connectives will be used by

the inference engine. This option is for experimentation with other logics. Obviously the map of local logics does not work with logics different from the standard of **Milord II**.

3. The last option is that of *Milord*. We can define the set of truth–values as fuzzy intervals and declare the conjunction by means of a function. In this case the other connectives and modus ponens are calculated by the system.

Notice that in all the cases we use the extension to intervals, including the logics used in *Milord*.

**Functions Declaration**

Users can define logical functions, or they can use the predefined functions of the system.

The user can define these functions by different means:

- Defining a *Truth Table* for the first and second case. In Figure 3.8 there is an example of conjunction table declaration.
- By means of a *S–expression* or using a predefined function in the last case.

| **Conjunction = Truth Table** | | | | |
|---|---|---|---|---|
| (( false | false | false | false | false ) |
| ( false | unlikely | unlikely | unlikely | unlikely ) |
| ( false | unlikely | may_be | may_be | may_be ) |
| ( false | unlikely | may_be | may_be | likely ) |
| ( false | unlikely | may_be | likely | true )) |

Figure 3.8: Truth table declaration for $T_{A_5}$.

The predefined functions which can be used are the well known:

- Łukasiewicz: $T(x, y) = max(0, x + y - 1)$
- Zadeh: $T(x, y) = min(x, y)$
- Probabilistic: $T(x, y) = xy$

### 3.4.3 Renaming

Because a module has its own logic, it should have a procedure to translate the certainty values used in its submodules. Then we introduce the construct *renaming* to make this translation.

```
Renaming:
    S/false ==> impossible
    S/unlikely ==> [impossible,few_possible]
    S/may_be ==> [few_possible,very_possible]
    S/likely ==> [very_possible,sure]
    S/true ==> sure
```

Figure 3.9: Renaming declaration example.

In Figure 3.9 there is an example of renaming declaration for the module *Gram* with the logic $B_7$ that has a submodule named *Gram_of_Sputum* that has the local logic $A_5$.

Finally we can include the complete declaration from the logic point of view of the module *Gram* (see the Figure 3.10).

## 3.5 Conclusions

In this Chapter we have introduced the local logics of modules of **Milord II**. Expertise implies to deal with imperfect information. The information managed by experts is imprecise and uncertain. A language for ESs must provide the possibility of expressing easily this kind of information. Furthermore the more adecuated language to express uncertainty is problem dependent.

**Milord II** introduces a family of multi–valued algebras that are useful to represent uncertainty by means of linguistic terms. The extension of these algebras to intervals of truth–values has been used to deal with imprecision and fuzzy sets.

Finally local logics has been introduced as a form to adapt the logic to the concrete problem and the method to allow the communication among modules with different logics has been provided.

```
Module Gram =
    Begin
    Module D= Respiratory_Diagnosis
    Module T= Type_of_Infection
    Module P= Previous_Treatment
    Module S= Gram_of_Sputum
    Export Pneumococcus, Haemophilus, Staphylococcus, Enterobacteria
    Deductive knowledge
        ...
        Truth values= (impos, few_p, sli_p, possib, quite_p, very_p, sure)
        Renaming
            D/false ==> impos
            D/true ==> sure
            T/false ==> impos
            T/true ==> sure
            P/impossible ==> impos
            P/sure ==> sure
            S/false ==> impos
            S/unlikely ==> [impos, sli_pos]
            S/may_be ==> possible
            S/likely ==> [quite_p, sure]
            S/true ==> sure
        Connectives:
            Conjunction = Truth Table
                ((impos, impos, impos, impos, impos, impos, impos)
                (impos, few_p, few_p, few_p, few_p, few_p, few_p)
                (impos, few_p, sli_p, sli_p, sli_p, sli_p, sli_p)
                (impos, few_p, sli_p, possib, possib, possib, possib)
                (impos, few_p, sli_p, possib, very_p, very_p, very_p)
                (impos, few_p, sli_p, possib, quite_p, very_p, very_p)
                (impos, few_p, sli_p, possib, quite_p, very_p, sure))
        end deductive
        ...    end
```

Figure 3.10: Example of logic declaration.

# Chapter 4

# Deduction by Specialization

We have seen in Chapter 2 how an application can be structured in modules. After that we have introduced in Chapter 3 the concept of local logics in modules. This allowed us to deal with different uncertainty and imprecision languages in different modules. Following the top–down description of **Milord II**, in this Chapter we will analyze the deductive knowledge of modules and its interpretation.

The deductive knowledge of **Milord II** is mainly composed of facts and rules as usual in Rule Based Systems . The particular kind of interpretation we give here to deductive knowledge, that is, our inference engine, is a key aspect of **Milord II**. We think that conventional inference engines produce a poor behavior of the ESs. We think of conventional inference engines as the well known backward and forward ones. In order to improve the behavior of ESs we propose an inference engine based on a new rule of inference called *specialization*. The deduction in **Milord II** is based on the specialization of KBs as we will see.

This Chapter is divided in three parts. The first one is devoted to the kind of behavior we want ESs to have, as the main motivation to introduce our concept of specialization of KBs. To do that we introduce informally the inference engine of **Milord II**. In the second part we present the logical foundation of the specialization calculus and the theoretical results about it. After that we will explain in detail the actual inference engine which is based on specialization. Finally we present all the extralogic components of deductive knowledge of **Milord II** introduced in the language to make easier the development of applications.

## 4.1 Enriched Behavior

The deductive knowledge component of the **Milord II** language is used by experts to represent the domain knowledge of their applications. Like *Milord*, **Milord II** is based on facts of order $0^+$ and production rules with uncertainty. They have proved to be useful in our application development experiences. For this reason further discussions on knowledge representation are avoided except

to motivate the introduction of new constructs.

In this Chapter we are interested in showing that the interpretation of the deductive knowledge plays an important role in the whole behavior of an ES, and that a good inference engine design can improve this behavior with respect to the one imposed by conventional inference engines.

At the moment, to simplify the explanation, we will focus the attention on the interpretation of facts and rules without the extra complications of control declarations and modular structuration[1].

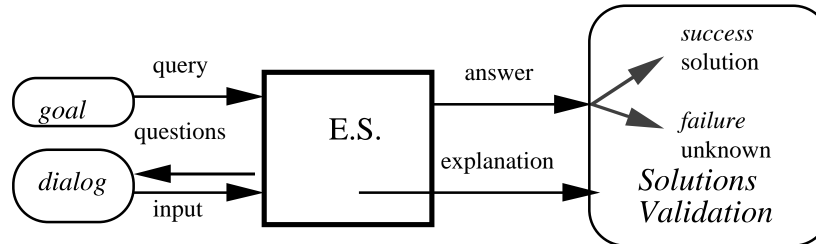


Figure 4.1: Standard Behavior of an ES.

In the Figure 4.1 there is represented the standard behavior of an ES. We will use that Figure to make clear which are the general aspects we are interested in. These are related to the communication of the ES with the user, the solutions generated by the ES and the validation of the ES.

**Communication:** First the user queries the system for the deduction of a fact. This fact will be the current goal of the ES[2] and the ES will try to find solutions for that fact. In order to obtain solutions, the ES stablishes a dialog with the user. The system asks questions to the user[3], raised by the process of deduction. We consider that the communication is the sequence of questions made to the user until a solution for the goal or a failure is found.

**Solutions:** The system makes inferences using the rules and the answers given by the user. Finally the system answers to the user with the solutions for the goal (if found), which is normally a truth–value in ESs with uncertainty. Furthermore the system gives some explanation of its answer. We consider that a solution is a pair composed by an answer and the explanation to that answer.

**Validation:** The aspects above allow experts to think in some kind of validation of their system. The simplest one is case validation, that is, to compare a

[1]Despite we abstract from modular structure, some aspects of the communication between an ES and the user can be extended to the communication between modules. In the text you will find references to modular structure when needed.

[2]We consider that the goals of the ES are facts. This corresponds to the interpretation of modules as objects able to answer facts that belongs to its export interface.

[3]In **Milord II** the facts asked to the user correspond to the imported facts of modules.

case (the answers given by the user to the ES questions, the goal and its solution by the user) with the corresponding solution obtained by the ES.

Communication, Solutions and Validation depend on the inference engine used. Conventional inference engines based on forward and backward strategies present a number of shortcomings in all three mentioned aspects of ES behavior. *Milord* was based on a backward inference engine with uncertainty. The architecture and behavior of the inference engine of **Milord II** have been designed to improve these three aspects of ES behavior.

Following these aspects we will analyze the insufficient behavior produced by conventional inference engines and explain what are the improvements on it we propose by means of an inference engine based on specialization. The first point considered is the architecture of **Milord II** inference engine. The second and third points are about one of the main topics of this thesis, that is, specialization.

### 4.1.1 Communication

In Chapter 1 we have explained that most of the applications developed with **Milord II** are interactive. Part of the problem solving behavior of the system consists in asking to the user the relevant information relative to the case to be solved. The user's confidence on the system is then highly related to the question–answer dialog he maintains with the ES. The user expects a sequence of questions which should be clearly related to the current goal. The questions asked and the order in which they are asked are very important to have a good interaction.

Conventional inference engines have search and deduction interleaved in the same process. For instance, in backward inference engines depth first and breath first strategies are part of the design of the inference engine. The search strategy of *Milord* was depth first. That means that an inference engine has a fixed search strategy. The search strategy is embedded in the inference engine and it can not be changed. Because the search strategy determines indirectly the questions asked and the order in which they are asked, it is difficult with these inference engines to obtain a good user interaction. The way to change it is usually by changing the order of rules, the order of the premises of the rules, and so on.

In this thesis we have not developed a theory of user system interaction. Here the very important point we consider is the architecture of **Milord II** inference engine which makes easier to have a good user interaction (see Figure 4.2). It is composed of two independent processes, that is, the search process and the deductive process.

**Search process:** Given a goal, the search process computes the information needed to reach the goal with maximum precision (comments on maximum precision and maximum certainty can be found in Section 3.2). In **Milord II** this process is independent of the deductive one. Classical inference engines are limited to the strategies implicitly implemented in the inference engine (depth first, breadth first, etc). This characteristic allows

us to implement different search strategies independently of the deductive process, including the conventional ones.

**Deductive process:** The inference engine of **Milord II** is based on specialization of KBs. Each new fact known will specialize the KB. The known facts have been previously selected by the search process. Then the current status of a KB is specialized with respect to the known information. Specialization will be explained along this Chapter.

The proposed architecture allows us to implement different search strategies depending on different criteria and independently of the deductive process. The search process is a part of the control of **Milord II**, and it will be explained in depth in Chapter 5. The deductive process is the main topic of the current Chapter.

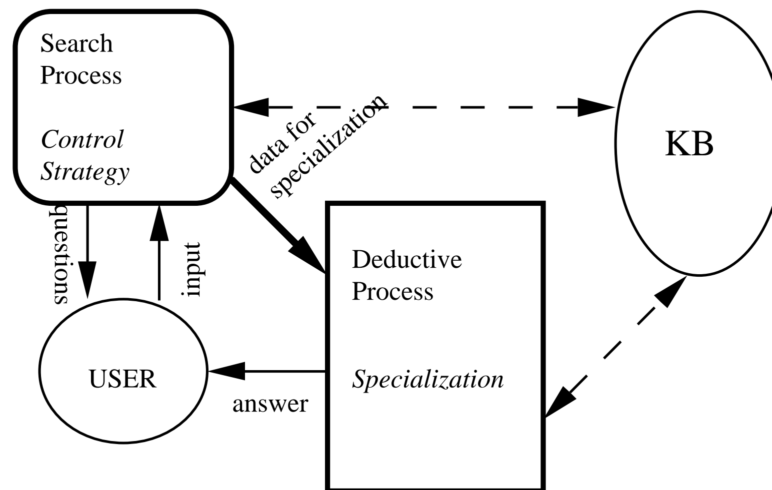


Figure 4.2: Inference Engine Architecture.

### 4.1.2 Solutions

A solution given to the user should be as much informative as possible, that is, the answer and its explanation should be clearly related to the previous question–answer interaction with the user. Any doubt on this relation causes the confidence in the system be on the decrease.

Real world ESs applications are very big and they demand a lot of interaction with the user. Sometimes the user does not know the answer to ES questions. There are different reasons for that. Maybe the user actually does not know the answer to the system question, or he does not know it at the time of the interaction. Frequently it can be too expensive to obtain that answer. For instance in medical environments, sometimes to answer a question implies to

produce an intrusive action on the patient. For all these reasons it is very important to be able to deal with incomplete information (lack of answers). Conventional inference engines are not able to work with incomplete information, they answer *unknown* to a goal if the system is not able to deduce the goal using the information given by the user.

Remember the standard ES behavior (please return on Figure 4.1). The user queries to the system whether a given fact can be deduced. If the system is able to deduce the fact, its certainty value is given back. Otherwise the answer is *unknown* (open world assumption). This behavior is rather poor because the system usually has much more information obtained implicitly in the process of deduction that could be useful to the user, for instance:

1. When the system is able to answer the user's query, the user might also be interested in knowing other deductive paths that would be useful to improve the solution, or to know other conditioned conclusions that could be deducible from this solution.

2. When the system is not able to answer a query, it gives back the value *unknown* maybe because the user did not provide enough information to the system. Thus, the communication would be much more informative if the system was able to answer, not *unknown*, but give the information the user should know to come up with a value for the query (this kind of answers are called *conditioned answers*).

All this information the ES has about the goal is usually not visible outside the system, and it could be used to better modelise communication among human experts. Looking carefully at how experts communicate their knowledge and at their problem solving procedures, we can find complex communication patterns. Sometimes experts cannot reduce their interaction only to the communication of certainty values for facts, that is, by giving a precise answer. For instance, in medical diagnosis, when experts communicate, they also need:

1. **To condition their decisions**. Suppose that it is not known whether a patient is allergic to penicillin. An expert considering the possibility of giving penicillin as treatment would say: *Penicillin is a good treatment from a clinical point of view provided that the patient has no allergy to penicillin.*

2. **To give suggestions that must be considered with solutions**. Experts usually give other suggestions (*antibiogram*) that are related to the solution (*pneumococcus*). For instance the expert might say: *Pneumococcus has been isolated in the culture of sputum. In this case it is strongly suggested to make an antibiogram to the patient.*

3. **To give conditioned suggestions to be considered together with decisions**. Another example of complex communication is the combination of the above two communication patterns: *Ciprofloxacine is a good*

*treatment, but if the patient is a woman on breast–feeding period she must stop breast–feeding*.

Specialization allows us to deal with incomplete knowledge giving as solutions conditioned answers, and producing complementary information as suggestions or recommendations. To model such communication protocols, we have to extend the ES answering procedure, by allowing to answer queries with sets of formulas (rules and facts). We propose the Specialization Calculus (Puyol et al., 1992b) as a type of deduction allowing to have all this enriched communication.

Let us introduce the specialization of KBs to better understand how we can obtain this kind of enriched solutions.

### Introduction to Specialization

Here we introduce informally the notion of specialization. In rule base systems, deduction is mainly based on the modus ponens:

$$A, A \rightarrow B \vdash B$$

Modus ponens is only applicable when every condition of the premise of the rule to be fired is satisfied, otherwise nothing can be inferred. We propose the use of partial evaluation to extract the maximum information even from incomplete knowledge about the truthvalue of the premises of a rule.

We base the partial evaluation of rules on the well known logical equivalence $(A \wedge B) \rightarrow C \equiv A \rightarrow (B \rightarrow C)$ which leads to the following boolean specialization inference rule:

$$A, A \wedge B \rightarrow C \vdash B \rightarrow C$$

The rule $B \rightarrow C$ is called the *specialization* of the rule $A \wedge B \rightarrow C$ with respect to the fact $A$. Notice that in the particular case of $B = \emptyset$, we recover the usual modus ponens rule.

It is easy to see that we can specialize rules deleting the known facts from the premise. Unknown facts remain as part of the premise of the rules. This leads to simpler rules that can be thought as compiled rules. For instance, suppose that we have the following rule:

**If** a **and** b **and** c **then conclude** d

Imagine that we only know $a$ and $c$ are true. Then the specialized rule is:

**If** b **then conclude** d

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

This boolean case can not be considered of great interest, but we can extend this specialization concept to the more interesting uncertainty calculus. For that we introduce the definition of what we call Specialization Inference Rule (SIR).

**Definition 4.1 (SIR)** *Given a fact A with certainty value* $\alpha$*, and a rule with certainty value* $\rho$*, then*

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

*where* $\rho' = MP^*_T(\alpha, \rho)$ *is the new truth value of the specialized rule*[4].

Now consider the following weighted rule:

**If** a **and** b **and** c **then conclude** d **is** *very_possible*

where *very_possible* is the truthvalue of the rule. Imagine we know that $a$ has the value *possible* and $c$ has the value *definite*. The resultant specialized rule could be:

**If** b **then conclude** d **is** *slightly_possible*

Notice that the truthvalue of the rule has changed because of the uncertain values of the facts $a$ and $c$. This is important when the KB contains a set of other rules that also deduce $d$, and all these rules are ordered by their precision. The specialized rule will change its place in the priority order affecting the corresponding effect in the search strategy as we will see in Section 5.1.2.

The specialization of a knowledge base consists on the exhaustive specialization of its rules. Rules whose conditions contain facts with known values are replaced by their specializations, in particular, rules that only have one known condition will be eliminated and its conclusion added to the KB as a new fact. This new fact will be used again to specialize the knowledge base. The process will finish when the knowledge base has no rule containing on its conditions a known fact.

Now we can return to the three examples on experts communication in the previous Section. We can translate the expert's statements into a set of formulas. It is easy to see that those formulas are a specialized part of a knowledge base related to a goal. For instance in the first example we can think of a knowledge base that contains the following rule:

**If no** (allergy) **and** ... **then conclude** penicillin_treatment **is** *good*

If the goal is *penicillin_treatment* then the rule selected is the above one. After that the system asks the questions related to the premise of this rule. Suppose all the conditions are satisfied except the one related to the fact *allergy* that is *unknown*. Finally the answer to the goal will be:

**If no** (allergy) **then conclude** penicillin_treatment **is** *good*

This rule expresses formally the answer given in the example. We can see that the system answers with a conditioned answer instead of saying that *penicillin_treatment is unknown*. It is very important to notice the following points about conditioned answers:

[4]SIR is parametric on the uncertainty propagation function $MP^*_T$ (modus ponens), particular for each uncertainty calculus.

- Conditioned answers is a form of answering a goal with part of the knowledge needed to reach it. Then we can say that the system communicates *knowledge* (rules) instead of only *data* (facts).

- The user can reconsider its answers to the questions of the system. In the example above, the user has answered *unknown* to the question *allergy*. The conditioned answer informs the user where *allergy* is used in order to obtain his goal. Then he can reconsider its answer trying to get this information.

Even in the case the system has reached the goal, it can also answer with complementary suggestions. Remember that the KB is specialized until it has no rule containing a know fact. Then we can obtain these suggestions by selecting those rules that have been specialized with the goal. Consider the two rules in the last example:

**If** ... **then conclude** ciprofloxacine **is** *good*
**If** ciprofloxacine **and** breast_feeding
**then conclude** stop_breast_feeding **is** *definite*

If the information of the case allows the system to conclude *ciprofloxacine*, the specialized rules will be:

*ciprofloxacine* **is** *good*
**If** breast_feeding **then conclude** stop_breast_feeding **is** *definite*

We can consider this result as an answer composed by the solution of the goal and a conditioned recommendation.

At this point let us give a new interpretation of the specialization of KBs. So far we have interpreted the specialization as a form to build answers by selecting part of a specialized KB. Now we will focus our attention on the inspection of a whole specialized KB.

A KB is programmed thinking in a concrete domain. For instance, the domain of a KB could be: *KB for treatment of pneumoniae acquired by adult patients outside the hospital environment*. This KB has no sense for problems out of this domain. In **Milord II** a KB is specialized with each information given by the user. We can interpret this specialization as a form of restricting the initial domain of the KB. Then, specialization goes from a KB in a domain to a KB in a more concrete domain. We can clarify this interpretation by means of an example.

In Figure 4.3 we can see an example of a general KB for pneumonia treatment. It is specialized for a case of women with gramnegative rods. We obtain a new KB for pneumonia treatment in the case of women with gramnegative rods. This interpretation of the specialization of KBs allows us to introduce how specialization can affect the validation of ESs.

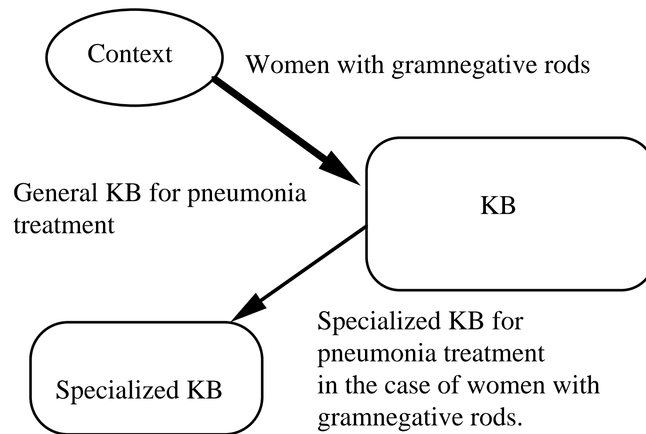


Figure 4.3: Example of specialization of a KB

### 4.1.3 Validation

Validation is a mandatory part of the development of ESs. Validation aims at checking that programs are free of errors and satisfy the user needs. A number of different methods and techniques exists for program validation. We can find an exhaustive study of validation and its new trends in (Meseguer, 1992).

Here we do not present any contribution to validation in the sense that we do not present a set of methods about how to validate an ES. We only want to show that specialization of KBs is a good technique to simplify the validation task, and we propose a very simple validation method based on specialization.

As shown in the last example, we can obtain specialized versions of an initial KB, producing a set of KBs in restricted domains. We can think in applying any validation method on these simpler specialized KBs. This can contribute to simplify the validation task similarly as argued for the modular structure in the introduction of Chapter 2.

The validation method used in **Milord II** is related to ES testing (the simplest case of validation). We define ES testing as the process of examining ES behavior by its execution on sample cases (test set) (Meseguer, 1992). The selection of the test set is an essential point for the testing process. This set should be large enough to be a representative sample of the program domain and yet small enough to allow the testing process to be executed on each element of the test set consuming a reasonable amount of resources. Testing has shown to be very effective in practice.

Normally the expert has a significative set of cases, for instance in medical diagnosis the cases are patient hospital records. After the execution of the cases in the ES, a comparative analysis between the results obtained from the ES an those given by human experts is done. This analysis of results and expected results next to simple explanations (rules fired, facts deduced, deductive paths, etc) helps the experts to detect errors and to do a fine tuning of the ES.

As we have explained above, conventional inference engines produce useful results when they have enought information to deduce the goal with a value different from *unknown*. Each element of the test set can be a large set of data. The expert should imagine how this set of data has produced the deduction of the result over a set of rules. A summary of the execution of the inference engine, like the fired rules, the evaluated facts, and so on, is not sufficient to imagine how the result has been deduced. This task can become more difficult when the number of rules of the system grows up. This kind of testing method involve the expert in the operational aspects of the system.

The testing method based on specialization avoids the experts enter into the operational aspects of the deduction. Specialization allows us to incrementally focus the KB in a concrete domain. The expert can detect errors or improve the KB by observing how a new information specializes his KB. After a specialization step the expert must agree with this new specialized version of the KB. He must agree with this version because it is the KB the expert would program for this restricted domain. Simple inspection of specialized KBs can give a simple method for the experts for validating their base without loss of the declarative interpretation of the language.

### 4.1.4 Summary

Finally we can summarize the most important characteristics of the inference engine architecture of **Milord II** and the differences with the inference engine of *Milord*. This differences are summarized in Table 4.1.

**Milord II** is based on an architecture with two independent processes (search process and deductive process) and on a mechanism of deduction based on specialization. This inference engine allows us to use any search strategy, allowing conditioned answers and simplified validation.

| | *Milord* | **Milord II** |
|---|---|---|
| inference | backward, forward | specialization |
| answer | atomic facts | formulas |
| search | implicit in the engine | independent process |
| goals | maximum certainty | maximum precision |

Table 4.1: Main differences between *Milord* and **Milord II** inference engines

The rest of this Chapter can be divided in two parts. The first one is related to the logical foundation of specialization and the design of the inference engine. For that, we introduce a simplified syntax and we develop the theory of specialization.

In the second part we introduce the real syntax and semantics of **Milord II** deductive knowledge. Deductive knowledge of **Milord II** is not only composed

of facts and rules. Practice in ES development has driven us to add a set of extralogical components to the base language.

## 4.2 Specialization Calculus

Specialization Calculus is the key point of the deductive process used in **Milord II**, and then of the whole behavior of the system. In this Section we give a formal description and study the properties of this calculus. Here we present the abstract syntax of the language, the semantics and a soundness theorem. Examples are written in real syntax, but they are easily understandable[5]. The purpose of the detailed examples is to introduce and acquire familiarity with the calculus with intervals of truthvalues.

### 4.2.1 Syntax

Here we present a very simplified syntax of facts and rules for the deductive knowledge of **Milord II**. It allows us to introduce formally the specialization calculus. Afterwards we will present the concrete syntax which does not introduce significant changes in the theoretical results.

A propositional language $\mathcal{L}_n = (A_n, \Sigma, \mathcal{C}, \mathcal{S}_n)$ is defined by:

- The set of linguistic terms $A_n$ as defined in Chapter 3.
- A signature $\Sigma$ consisting on a set of atomic propositional symbols plus *true* and *false*.
- A set of Connectives: $\mathcal{C} = \{\neg, \wedge, \rightarrow\}$
- A set of Sentences: $\mathcal{S}_n$ = Mv–Literals $\cup$ Mv–Rules

  Sentences are pairs of classical–like propositional sentences and intervals of truth–values. The classical–like propositional sentences are restricted to be literals or rules. Thus, the sentences of the language are of the following types:

  **Mv–Atoms:** $\{(p, V) \mid p \in \Sigma$ and $V \in Int(A_n)\}$

  **Mv–Literals:** $\{(p, V) \mid (p, V) \in$ Mv–Atoms or $p = \neg q$ and $(q, V) \in$ Mv–Atoms$\}$

  **Mv–Rules:** $\{(p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q, V) \mid p_i$ and $q$ are literals, $V = [a, 1]$ is an upper interval of $Int(A_n)$ where $a > 0$, and $\forall i, j (p_i \neq p_j, p_i \neq \neg p_j, q \neq p_j, q \neq \neg p_j)\}$

[5] The logic used in the examples of this chapter is the one defined in Chapter 3. Remember that the linguistic terms are $A_5$ = {impossible, slightly_possible, possible, very_possible, definite}.

This syntax is close to the standard one in ESs, that is, facts and production rules. The most important difference is that facts and rules are weighted by intervals of truthvalues. The atomic symbols of the signature are what we call facts in ES terminology. Rules are composed of a set of conditions and a conclusion. The conditions and the conclusion are facts or negations of facts. As usual, the facts contained in a rule only appear once, in the conditions or in the conclusion.

In the next sections we will introduce the real syntax of facts and rules. Here we limit our expansion to the logical components of deductive knowledge and we will use the terminology introduced above.

The connectives are translated to the real syntax as: *no* ($\neg$), *and* ($\wedge$), and *if* $\cdots$ *then conclude* ($\rightarrow$). An example of rule in **Milord II** syntax is:

R005 **If** macrol **and no** (light_seriousness) **and**
entered_to_hospital **then conclude no** (roxi) **is** *very_possible*

Notice that the symbol *very_possible* after *is* represents the truthvalue of the mv–rule. Only an element of $A_n$ is introduced because the truthvalue of the mv–rule is always implicitly considered an interval from an element of $A_n$ to 1. We could say that this value represents the more pessimistic value of the rule. It is easy to see that the translation of the previous rule to the abstract syntax is:

(macrol $\wedge$ $\neg$light_seriousness $\wedge$ entered_to_hospital
$\rightarrow$ $\neg$roxi,[*very_possible*,1])

There are two reasons to consider this kind of intervals for the conclusions of rules. The first one is for simplicity. Despite the utility of using intervals of truthvalues, experts express better their knowledge using only one value for the rules. The second one is related to the semantics of specialization and it will be explained in the next Section.

### 4.2.2 Semantics

After the description of the syntax of the abstract language, we present the meaning of the sentences introduced above. This allows us to give an interpretation of facts and rules with no ambiguity. It is very easy to missinterpret the language as often do experts by translating the rules into natural language and incorporating all the ambiguities it has. Then it is mandatory the reference to the exact meaning of the language.

#### Models

Models $M_\rho$ are defined by valuations $\rho$, i.e. mappings from the first components of sentences to $A_n$. The elementary case is the valuation of atomic symbols. For instance we can say that the fact *macrol* has the value *possible*, that is, $\rho(macrol)$ = *possible*. From this elementary valuation and the operators $N_n$ (negation),

$T$ (conjunction) and $I_T$ (implication) presented in Section 3.1 we can obtain valuations of non atomic sentences.

Valuations of non atomic sentences as defined below hold that:

**M1:** $\rho(\neg p) = N_n(\rho(p))$

**M2:** $\rho(p_1 \wedge p_2) = T(\rho(p_1), \rho(p_2))$

**M3:** $\rho(p \rightarrow q) = I_T(\rho(p), \rho(q))$

**M4:** $\rho(true) = 1$

**M5:** $\rho(false) = 0$

We can build an example with the facts and the rule given in the syntax example. Suppose that our model $M_\rho$ is:

$\rho$(macrol) = *possible*
$\rho$(light_seriousness) = *impossible*
$\rho$(entered_to_hospital) = *definite*
$\rho$(roxi) = *slightly_possible*

We can extend the valuation function to the example rule as follows:

$\rho$(macrol ∧ ¬light_seriousness ∧ entered_to_hospital → ¬roxi) =
$I_T(T(\rho(\text{macrol}), T(N_n(\rho(\text{light_seriousness})), \rho(\text{entered_to_hospital}))),$
$N_n(\rho(\text{roxi}))) =$
$I_T(T(\textit{possible}, T(N_n(\textit{false}), \textit{true})), N_n(\textit{slightly_possible})) = \textit{definite}$

### Satisfaction Relation

Given a model, that is, the valuations of facts and rules, we define when a sentence is satisfied by a model. The satisfaction relation is defined as follows:

**Definition 4.2 (Satisfaction Relation)** *The Satisfaction Relation between models and sentences is defined by:*

$$M_\rho \models (p, V) \textit{ iff } \rho(p) \in V$$

*where* $V \in Int(A_n)$.

The satisfaction of a set of sentences is the satisfaction of each of them. This satisfaction relation introduces in our system the notion of imprecision presented in Chapter 3, a sentence is satisfied if the valuation of the proposition belongs to an interval of truthvalues (it is not necessarily equal to a truthvalue). For instance we can show that the model above satisfies the rule given in the syntax example:

{$\rho$(macrol) = *possible*, $\rho$(light_seriousness) = *impossible*,
$\rho$(entered_to_hospital) = *definite*, $\rho$(roxi) = *slightly_possible*} ⊨
(macrol ∧ ¬light_seriousness ∧ entered_to_hospital
→ ¬roxi,[*very_possible*,1])

It is easy to see that:

$\rho$(macrol $\wedge$ $\neg$light_seriousness $\wedge$ entered_to_hospital $\rightarrow$ $\neg$roxi) = *definite* $\in$ [*very_possible*,1]

**Semantical Entailment**

From the definitions of models and satisfaction relation given above, we can introduce in the usual way the notion of semantical deduction, that is, when a sentence is semantically deducible from a set of sentences.

**Definition 4.3 (Semantical Entailment)** *Semantical entailment between sets of sentences and sentences is defined as usual:*

$\Gamma \models A$ *iff for any model* $M_\rho \models \Gamma$ *implies* $M_\rho \models A$*,*

*for any set of sentences* $\Gamma$ *and sentence* $A$*.*

Following the same example consider that $\Gamma$ and $A$ are:

$\Gamma$ = {(macrol,[*possible*,*definite*]),
(light_seriousness,[*impossible*,*impossible*]),
(entered_to_hospital,[*definite*,*definite*]),
(macrol $\wedge$ $\neg$light_seriousness $\wedge$ entered_to_hospital
$\rightarrow$ $\neg$roxi,[*very_possible*,*1*])}

$A$ = (roxi,[*impossible*,*possible*])

Now we want to prove that $\Gamma \models A$. First of all we should find all the models that satisfies $\Gamma$. Using the satisfaction relation it is easy to see that the valuations of the atoms must be :

$$\rho(macrol) = \begin{cases} possible \\ very_possible \\ definite \end{cases}$$

$\rho$(light_seriousness) = *impossible*
$\rho$(entered_to_hospital) = *definite*

$$\rho(roxi) = \begin{cases} impossible \\ slightly_possible \\ possible \\ very_possible \\ definite \end{cases}$$

From these atomic valuations which combinations define fifteen possible models, we can compute which are the valid models that satisfy the rule:

$\rho$(macrol $\wedge$ $\neg$light_seriousness $\wedge$ entered_to_hospital $\rightarrow$ $\neg$roxi) =
$I_T(T(\rho(\text{macrol}), T(N_n(\rho(\text{light_seriousness})), \rho(\text{entered_to_hospital}))),$
$N_n(\rho(\text{roxi}))) \in [\textit{very_possible}, \textit{1}]$

We can simplify the expression with the facts *light_seriousness* and *entered_to_hospital* that only can take a boolean value:

$$I_T(\rho(\text{macrol}), N_n(\rho(\text{roxi}))) \in [\textit{very_possible}, \textit{1}]$$

Using the Table 3.3 of the operator $I_T$ in this example, we can build the Table 4.2 that contains the valid models obtained.

| $\rho$(light_seriousness) | $\rho$(entered_to_hospital) | $\rho$(macrol) | $\rho$(roxi) |
|---|---|---|---|
| impossible | definite | possible | impossible |
| | | | sli_possible |
| | | | possible |
| | | very_possible | impossible |
| | | | sli_possible |
| | | | possible |
| | | definite | impossible |
| | | | sli_possible |

Table 4.2: Valid models of the example.

Then we have obtained eight models that satisfy $\Gamma$. It is easy to see that all the valuations of *roxi* are in the interval [*impossible*,*possible*], so $\Gamma$ entails $A$.

Now we are interested in semantical deduction. From mv–facts and mv–rules we want to obtain new mv–facts and specialized mv–rules. A set of interesting properties of the semantic entailment that will play a major role in later proofs is presented next.

**Proposition 4.1** *If $p, q, p_1, \ldots, p_n$ denote literal symbols then the following properties are fulfilled:*

**SR1:** $(p, V) \models (p, W) \Leftrightarrow V \subseteq W$

**SR2:** $(p, V) \models (\neg p, W) \Leftrightarrow N_n^*(V) \subseteq W$

**SR3:** $(p, V), (p, W) \models (p, U) \Leftrightarrow V \cap W \subseteq U$

**SR4:** $(p_i, V_i), (p_1 \wedge \cdots \wedge p_n \rightarrow q, V) \models (p_1 \wedge \cdots \wedge p_{i-1} \wedge p_{i+1} \wedge \cdots \wedge p_n \rightarrow q, W) \Leftrightarrow MP_T^*(V_i, V) \subseteq W$

**SR5:** $MP_T^*(T^*(V_1, \ldots, V_n), W) = MP_T^*(V_1, MP_T^*(V_2, \ldots, MP_T^*(V_n, W) \ldots))$, *if* $W = [w, 1]$

These properties[6] give semantics to the negation of mv–atoms and the specialization of mv–rules. The property **SR5** is the justification of the equivalence which is the base of specialization ($a \wedge b \rightarrow c \equiv a \rightarrow (b \rightarrow c)$). This is the other reason cited above to use upper intervals in the mv–rules.

[6]The proof of these properties can be found in (Puyol et al., 1992c) and Appendix B.

### 4.2.3 Specialization Calculus

After the semantics we define the syntactical deduction that will be the base of the inference engine and we prove that it is sound. The specialization calculus is based on the following axioms and inference rules:

1. Axioms:

   **A1:** $(true, [1, 1])$

   **A2:** $(false, [0, 0])$

2. Axiom schemes:

   **AS1:** $(p, [0, 1])$

3. Inference rules:

   **Weakening:** $(p, V_1) \vdash (p, V_2)$ where $V_1 \subseteq V_2$, for any literal $p$

   **Not–introduction:** $(p, V) \vdash (\neg p, N_n^*(V))$, for any atom $p$
   $(\neg p, V) \vdash (p, N_n^*(V))$, for any atom $p$

   **Composition:** $(p, V_1), (p, V_2) \vdash (p, V_1 \cap V_2)$, for any literal $p$

   **SIR:** $(p_i, V_i), (p_1 \wedge \cdots \wedge p_i \wedge \cdots \wedge pn \rightarrow q, V_r) \vdash$
   $(p_1 \wedge \cdots \wedge p_{i-1} \wedge p_{i+1} \wedge \cdots \wedge p_n \rightarrow q, MP_T^*(V_i, V_r))$, for any literals $p_1 \cdots p_n$ and $q$

It is easy to see that these inference rules are deduced from the properties SR1–SR4. This is the simplest system for our concept of specialization of KBs. We want to obtain mv–facts and specialized mv–rules. We are not interested in other kind of rules. Finally we can give an example of deduction. Consider this initial set of sentences composed by the above rule *R005* and concrete values for the facts *macrol*, *light_seriousness* and *entered_to_hospital*.

(macrol,[*possible*,*definite*])
(light_seriousness,[*impossible*,*impossible*])
(entered_to_hospital,[*definite*,*definite*])
(macrol ∧ ¬light_seriousness ∧ entered_to_hospital
→ ¬roxi,[*very_possible*,*1*])

Using Not–introduction and SIR inference rules we can deduce syntactically the following set of sentences[7]:

(macrol,[*possible*,*definite*])
(¬macrol,[*impossible*,*possible*])
(light_seriousness,[*impossible*,*impossible*])
(¬light_seriousness,[*definite*,*definite*])

[7]We do not apply the Weakening inference rule for the sake of simplicity. This rule would generate new more imprecise mv–atoms. These mv–atoms would specialize again the mv–rules producing new more imprecise mv–rules.

(entered_to_hospital,[*definite*,*definite*])
($\neg$entered_to_hospital,[*impossible*,*impossible*])
(macrol $\wedge$ $\neg$light_seriousness $\wedge$ entered_to_hospital
$\rightarrow$ $\neg$roxi,[*very_possible*,*1*])
($\neg$light_seriousness $\wedge$ entered_to_hospital $\rightarrow$ $\neg$roxi,[*possible*,*1*]))
(entered_to_hospital $\rightarrow$ $\neg$roxi,[*possible*,*1*]))
($\neg$roxi,[*possible*,*1*])
(roxi,[*impossible*,*possible*])

For simplicity we have not written all the resultant sentences obtained from the application of weakening. Finally the conclusion is:

$A =$ (roxi,[*impossible*,*possible*])

### 4.2.4 Soundness and Completeness

From properties SR1, SR2, SR3 and SR4 of the semantical entailment, it is easy to check that the above specialization calculus is sound.

**Theorem 4.1 (Soundness)** *Let $A$ be a sentence and $\Gamma$ a set of sentences. Then $\Gamma \vdash A$ implies $\Gamma \models A$*

It is straightforward to see that our deductive system is not complete. For instance, we have $\{(p \rightarrow q, 1), (q \rightarrow r, 1)\} \models (p \rightarrow r, 1)$ but $\{(p \rightarrow q, 1), (q \rightarrow r, 1)\} \not\vdash (p \rightarrow r, 1)$. It is also the case that the language is not complete for literal deduction in general. For instance, we have $\{(p \rightarrow q, 1), (\neg p \rightarrow q, 1)\} \models (q, 1)$ but $\{(p \rightarrow q, 1), (\neg p \rightarrow q, 1)\} \not\vdash (q, 1)$. However, it can be proved that the system is complete for literal deduction in the context of a restricted language setting. Soundness and restricted completeness are proved in (Puyol et al., 1992c) and in Appendix B.

Despite our deductive system is not complete, we can show that it is useful for our purpose of obtaining specialization of KBs. We need to specialize a rule when we know the truthvalue of an atom that belongs to the premise of that rule and in this case we have:

$$\{(p_1, V_1), (p_1 \wedge \cdots \wedge p_n \rightarrow q, W)\} \models (p_2 \wedge \cdots \wedge p_n \rightarrow q, U) \Leftrightarrow$$
$$\{(p_1, V_1), (p_1 \wedge \cdots \wedge p_n \rightarrow q, W)\} \vdash (p_2 \wedge \cdots \wedge p_n \rightarrow q, U)$$

To finish this part we introduce the implementation of that Specialization Calculus.

## 4.3 Implementation

In the above Section we have introduced the Specialization Calculus used in **Milord II**. Axioms and inference rules are the sufficient mechanisms to do specialization from the logic point of view. We have proved that the properties we were interested in hold. Now we want to implement that Specialization Calculus.

The requirements for the inference engine are closely related to the behavior of the whole system. Remember that the specialization of deductive knowledge is the base for users/system communication. Furthermore the inference engine must also handle the control knowledge. In summary, the inference engine is a key part of the whole system **Milord II**. In this Section we will present the requirements of the inference engine and the concrete implementation.

### 4.3.1 Inference Engine Design

The simplest inference engine would be an inference engine that given a KB it would apply all the inference rules of the Specialization Calculus until there is not any rule that can be applied. In the last example we have showed this kind of behavior. But when we design the inference engine of **Milord II** we have new requirements that determine which and when rules have to be applied.

In order to preserve the correctness of the inference engine with respect to the semantics of Specialization Calculus, the inference engine does not introduce here extralogical components. The axioms and inference rules presented above are the only mechanisms to do specialization.

As usual we are interested in designing an efficient program but we do not make a special point of this. Despite this we have introduced some control to improve the efficiency of specialization.

Furthermore we should take into account that this inference engine must be integrated in the whole system. Possible simplifications of the inference engine are not possible because we need to handle the actions of the local control of a module (control will be explained in Chapter 5). The following are the more important points that we take into account in the design of the inference engine of **Milord II**:

- The task of the inference engine is to produce specialization of deductive knowledge. As seen in Section 4.1 the specialized deductive knowledge is used to present an enriched behavior of ESs. The internal representation of deductive knowledge should facilitate that task. We have chosen an internal representation that it is close to our concept of specialization.

- When deducing facts we are only interested in values (intervals of truth–values) having maximum precision. Then the inference engine will assign a definitive value to a fact when it is the most precise (you can find comments on imprecision and its control in Sections 3.2.2 and 5.1.2 respectively) that we can obtain from the deductive knowledge in the current case; otherwise the values are considered to be provisional.

- The inference engine must be interleaved with control. Then it should fit the actions made by the control. We avoid some simplifications that would produce a more efficient inference engine because in the inference engine design we must foresee the actions of the control.

Following the above criteria of design now we will comment the use of the inference rules of the Specialization Calculus.

The first point is that we want to obtain maximum precision. This implies that the inference engine must not use the Weakening inference rule that produces valid but less precise values. Another consequence of maximum precision is that the system specializes the mv–rules with the more precise mv–atoms. Then, it is not necessary to conserve the previous versions of the mv–rules because the rules will not be specialized again with the same mv–atoms. This implies the substitution of mv–rules with their specialized versions.

From the expert point of view a KB is represented by means of mv–rules and mv–atoms. Specialization of a KB should conserve that representation as shown in the introduction of this Chapter. Then we are not interested in creating mv–literals. Not–Introduction is only used when necessary to calculate the new truthvalue of mv–rules or that of their conclusion when they are specialized.

Finally notice that Composition is used to calculate the truthvalue of a fact when more that one rule conclude it. Each rule gives a provisional value to the conclusion. Finally the definitive value is obtained by means of Composition.

Following this requirements we will define the inference engine. We will use functional and algorithmic descriptions. We use a simple example to illustrate the behavior of the inference engine. First of all we introduce the internal data structure of a KB.

### 4.3.2 Internal Representation of Deductive Knowledge

Here we present the internal representation of deductive knowledge to make clear what is the task of the inference engine and what are the entries to control.

We are not interested in maintaining a data structure composed of facts and rules. We propose a new data representation that it is very close to the real implementation and that clarifies control and extralogical components of the inference engine.

We maintain the above definitions for mv–atoms, and mv–literals, but we propose a new representation of mv–rules that allows us to make easy the functional descriptions by means of set operations.

**Definition 4.4 (Mv–Rule)** *A mv–rule is represented as a 3–tuple* $r = (m_r, c_r, \rho_r)$ *where* $m_r$ *is the premise (a set of literals),* $c_r$ *is the conclusion (a literal) and* $\rho_r$ *is the truthvalue of the rule (an interval of truthvalues such that* $\rho_r = [\alpha, 1]$ *and* $\alpha \in A_n$*).*

For instance the mv–rule (c $\wedge$ d $\rightarrow$ e,[$\rho_2$,1]) can be represented as the 3–tuple: $(\{c, d\}, e, [\rho_2, 1])$. From that we can give the definition of a knowledge base. Given a set of mv–rules, it consists in associating each atom appearing in the mv–rules with its current truth–value (provisional or definitive one) and the mv–rules that can deduce it, that is, those whose conclusion is that atom or the negation of that atom. In the following we will call knowledge base the internal representation of the deductive knowledge.

**Definition 4.5 (Knowledge Base)** *Let* $R$ *be a set of mv–rules and let* $F_R$ *be the set of all atoms appearing in the rules of* $R$*. We define a knowledge base*

*$KB_R$ as a mapping*[8]*:*

$$KB_R : F_R \to Int(A_n) \times \mathcal{P}(R)$$

*where, for each $f \in F_R$, $KB_R(f) = (v_f, R_f)$, being $R_f = \{r \in R | r = (m_r, c_r, \rho_r)$ and $c_r = f$ or $c_r = \neg f\}$*

Initially we build a knowledge base with truthvalues of the atoms $[0, 1]$. It means that the atoms initially has the most imprecise value. Notice that a KB with all the atoms with truthvalues $[0, 1]$ is always consistent in our specialization calculus (axiom scheme **AS2**). The truthvalues of atoms can be changed by the rules that deduce them (giving a more precise truthvalue by means of Composition inference rule) or by external valuations of atoms. The truthvalue of an atom is considered to be provisional when there are rules that can conclude it; otherwise it is considered to be definitive.

Now we can see an example of initial KB. Suppose that we have a deductive knowledge composed of the following set of mv–rules:

$R = \{$(a $\wedge$ b $\to$ c,$[\rho_1$,1]), (a $\wedge$ f $\to$ $\neg$ c,$[\rho_2$,1]), (c $\wedge$ d $\to$ e,$[\rho_3$,1])$\}$

It is easy to see that the set $F_R$ is:

$F_R = \{$(a, b, c, d, e, f)$\}$

Now we can translate the example to the new representation. We create a table (mapping) from atoms of $F_R$ to its initial truthvalue ([0,1]) and the set of mv–rules that deduce them. Then the knowledge base is the mapping $KB$ such that:

$KB(a) = ([0, 1], \emptyset)$
$KB(b) = ([0, 1], \emptyset)$
$KB(c) = ([0, 1], \{(\{a, b\}, c, [\rho_1, 1]), (\{a, f\}, \neg c, [\rho_2, 1])\})$
$KB(d) = ([0, 1], \emptyset)$
$KB(e) = ([0, 1], \{(\{c, d\}, e, [\rho_3, 1])\})$
$KB(f) = ([0, 1], \emptyset)$

Now we will define the specialization of KBs represented by the above formalism. We will use the same example to illustrate the explanation.

### 4.3.3 Specialization

First of all we give a functional description of specialization of mv–rules. Giving a mv–rule and a mv–atom, the mapping $\mathcal{S}_{\mathcal{R}}$ will specialize the mv–rule with respect of that mv–atom. It will return the new specialized mv–rule, or a new mv–atom in the case that the initial mv–rule would be able to deduce its conclusion.

[8] $\mathcal{P}(R)$ is the powerset of the set $R$.

**Definition 4.6 (Specialization of Mv–Rules)** *Let $R$ be a set of mv–rules and $F^*$ a set of mv–atoms $F^* = \{(p, \rho_p) | p \in F_R\}$. We define $\mathcal{S}_\mathcal{R}$ as a mapping:*

$$\mathcal{S}_\mathcal{R} : R \times F^* \to R \times F^*$$

$$\mathcal{S}_\mathcal{R}(r, (p, \rho_p)) = \begin{cases} (r, (\emptyset, [0,1])) & \text{if } p \notin m_r \text{ and } \neg p \notin m_r \\ (r', (\emptyset, [0,1])) & \text{if } p \in m_r \text{ or } \neg p \in m_r \\ (\emptyset, (q, \alpha)) & \text{if } m_r = \{p\} \text{ or } m_r = \{\neg p\} \\ & \quad \text{and } c_r = q \text{ or } c_r = \neg q \end{cases}$$

*where*

$$r' = \begin{cases} (m_r - \{p\}, c_r, MP^*_T(\rho_p, \rho_r)) & \text{if } p \in m_r \\ (m_r - \{\neg p\}, c_r, MP^*_T(N^*_n(\rho_p), \rho_r)) & \text{if } \neg p \in m_r \end{cases}$$

*and*

$$\alpha = \begin{cases} MP^*_T(\rho_p, \rho_r) & \text{if } m_r = \{p\} \text{ and } c_r = q \\ N^*_n(MP^*_T(\rho_p, \rho_r)) & \text{if } m_r = \{\neg p\} \text{ and } c_r = q \\ MP^*_T(N^*_n(\rho_p), \rho_r) & \text{if } m_r = \{p\} \text{ and } c_r = \neg q \\ N^*_n(MP^*_T(N^*_n(\rho_p), \rho_r)) & \text{if } m_r = \{\neg p\} \text{ and } c_r = \neg q \end{cases}$$

Negations of mv–atoms are calculated by means of the Not–introduction inference rule, and the specialization of mv–rules is done using SIR inference rule. Notice that we only generate mv–atoms. Taking again the same example we can do the specialization of the mv–rule of the last example obtaining a new specialized mv–rule:

$\mathcal{S}_\mathcal{R}$(({a, f}, ¬ c,[$\rho_2$,1]), (f,[$\rho_4, \rho'_4$]) =
(({a}, ¬ c,[$MP_T(\rho_4, \rho_2)$,1]), ∅)

If we specialize again the rule we obtain a mv–atom:

$\mathcal{S}_\mathcal{R}$(({a}, ¬ c,[$MP_T(\rho_4, \rho_2)$,1]), (a,[$\rho_5, \rho'_5$]) =
(∅, (c, $N^*_n(MP^*_T([MP_T(\rho_4, \rho_2), 1], [\rho_5, \rho'_5]))$)))

After the definition of rule specialization we can explain atom specialization. A kb–atom is a structure that associates to an atom its truthvalue and the rules that conclude it. Then, atom specialization will modify the structure of a kb–atom with a mv–atom.

First of all we find a mv–rule from the set of mv–rules that conclude the atom such that it can be specialized with respect to that mv–atom. If no rule of this type can be found the specialization of the kb–atom returns the same structure. Otherwise it returns a new specialized rule ($r' \neq r$) or a new mv–atom ($r = \emptyset$). If the specialization of a rule returns a new rule ($\mathcal{S}_\mathcal{R}(r, f^*) = (r', (\emptyset, [0,1]))$) then we substitute the rule by the specialized one and the truthvalue is not changed. Notice that $[a,b] \cap [0,1] = [a,b]$. If it returns a new mv–atom ($\mathcal{S}_\mathcal{R}(r, f^*) = (\emptyset, (p', v'))$) the rule is eliminated and the new truthvalue is calculated by means of the Composition inference rule.

**Definition 4.7 (Atom Specialization)** *Let $R$ a set of mv–rules and $F^*$ a set of mv–atoms, the specialization of facts is a mapping such that:*

$$\mathcal{S}_\mathcal{C} : Int(A_n) \times \mathcal{P}(R) \times F^* \rightarrow Int(A_n) \times \mathcal{P}(R)$$

$$\mathcal{S}_\mathcal{C}((v, R_f), f^*) \\ = \begin{cases} \mathcal{S}_\mathcal{C}((v \cap v', R_f - \{r\} + \{r'\}), f^*) & \textit{(*)} \\ (v, R_f) & \textit{otherwise} \end{cases}$$

*(*) such that $\mathcal{S}_\mathcal{R}(r, f^*) = (r', (p', v'))$ and $r' \neq r$*

*being $R_f = \{r \in R | r = (m_r, c_r, \rho_r)$ and $c_r = f$ or $c_r = \neg f\}$*

For instance, suppose we have the following kb–atom:

$$KB(c) = ([0, 1], \{(\{a, b\}, c, [\rho_1, 1]), (\{a, f\}, \neg c, [\rho_2, 1])\})$$

We want to specialize it with the mv–atom $f^* = (a, [\rho_5, \rho'_5])$. Then the specialization is:

$$\mathcal{S}_\mathcal{C}((v, R), f^*) = \\ \mathcal{S}_\mathcal{C}(([0, 1], \{(\{a, b\}, c, [\rho_1, 1]), (\{a, f\}, \neg c, [\rho_2, 1])\}), (a, [\rho_5, \rho'_5])) = \\ ([0, 1], \{(\{b\}, c, [\rho_6, 1]), (\{f\}, \neg c, [\rho_7, 1])\})$$

because the specialization of the rule is:

$$\mathcal{S}_\mathcal{R}((\{a, f\}, \neg c, [\rho_2, 1]), (a, [\rho_5, \rho'_5])) = ((\{f\}, \neg c, [\rho_6, 1]), \emptyset) \\ \mathcal{S}_\mathcal{R}((\{a, b\}, c, [\rho_1, 1]), (a, [\rho_5, \rho'_5])) = ((\{b\}, c, [\rho_7, 1]), \emptyset)$$

We want to specialize this resultant kb–atom with the mv–atom $f^* = (f, [\rho_4, \rho'_4])$. Then

$$\mathcal{S}_\mathcal{C}((v, R), f^*) = \\ \mathcal{S}_\mathcal{C}(([0, 1], \{(\{b\}, c, [\rho_6, 1]), (\{f\}, \neg c, [\rho_7, 1])\}), (f, [\rho_4, \rho'_4])) = \\ ([0, 1] \cap [0, \rho_8], \{(\{b\}, c, [\rho_6, 1])\})$$

because the specialization of the rule is:

$$\mathcal{S}_\mathcal{R}((\{f\}, \neg c, [\rho_3, 1]), (f, [\rho_4, \rho'_4])) = (\emptyset, \rho_8)$$

It is easy to see that:

$$KB(c) = ([0, \rho_8], \{(\{b\}, c, [\rho_6, 1])\})$$

We can interpret that the atom $c$ has provisional value $[0, \rho_8]$ because there is a mv–rule that conclude that atom yet. Finally we can arrive to the whole knowledge base specialization:

Given a set of mv–atoms to specialize a KB, each mv–atom is used to specialize the KB. This process of specialization will produce new mv–atoms. These mv–atoms will be used to specialize the KB again. Specialization stops when there are no new mv–atoms to specialize the KB.

**Definition 4.8 (KB Specialization)** *Let $KB$ be a set of knowledge bases and $F^*$ a set of mv–atoms such that $F^* = \{(p, V)|p \in F_R\}$. KB specialization is defined as a mapping:*

$$\mathcal{S_{KB}} : KB \times \mathcal{P}(F^*) \rightarrow KB$$

$$\mathcal{S_{KB}}(kb, F^*) = \begin{cases} \mathcal{S_{KB}}(kb', F^* - \{f^*\} + F^{*'}) & \text{if } F^* \neq \emptyset \\ kb & \text{otherwise} \end{cases}$$

*where* $\forall f_{kb} \in F_R$,

$$kb'(f_{kb}) = \begin{cases} (v, \emptyset) & \text{if } (f_{kb}, v) = f^* \text{ or } kb(f_{kb}) = (v, \emptyset) \\ \mathcal{S_C}(kb(f_{kb}), f^*) & \text{otherwise} \end{cases}$$

*and*

$$F^{*'} = \{y = (p, \rho_y)|kb(p) = (v, R), \mathcal{S_C}(kb(p), f^*) = (\rho_y, \emptyset), p \in F_R\}$$

Finally we can see the last example consisting in specializing the last KB with the atom $b$.

$$\mathcal{S_{KB}}(kb, \{(b, \rho_{10})\}) = \mathcal{S_{KB}}(kb', F^*)$$

where $f^* = (b, \rho_{10})$

$KB'(a) = (\rho_5, \emptyset)$
$KB'(b) = ([0, 1], \emptyset)$
$KB'(c) = \mathcal{S_C}((\rho_9, \{(\{b\}, c, [\rho_1, 1])\}), (b, \rho_{10})) = (\rho_{11}, \emptyset)\})$
$KB'(d) = ([0, 1], \emptyset)$
$KB'(e) = \mathcal{S_C}(([0, 1], \{(\{c, d\}, e, [\rho_2, 1])\}), (b, \rho_{10})) =$
$([0, 1], \{(\{c, d\}, e, [\rho_2, 1])\})$
$KB'(f) = (\rho_7, \emptyset)$

$$F^* = \{(c, \rho_{11})\}$$

Finally

$KB''(a) = (\rho_5, \emptyset)$
$KB''(b) = ([0, 1], \emptyset)$
$KB''(c) = (\rho_{11}, \emptyset)$
$KB''(d) = ([0, 1], \emptyset)$
$KB''(e) = ([0, 1], \{(\{d\}, e, [\rho_{12}, 1])\})$
$KB''(f) = (\rho_7, \emptyset)$

We have seen the specialization calculus and the concrete implementation by the above description of the inference engine function $\mathcal{S_{KB}}$. This inference engine has been designed avoiding extralogical components and then assuring the correctness of the inference engine with respect to the semantics of specialization calculus. In Chapter 5 we will enrich the inference engine with the actions of control.

## 4.4 The Deductive Knowledge Language

This Section starts a different approach to **Milord II** from that seen above. In the current Chapter we have explained in a formal way a propositional language close to that of **Milord II** and its interpretation. We have introduced the basic logical syntax, the semantics, the properties of this language and the inference engine that interprets it. Here we introduce the real language of **Milord II** which has been build on top of the previous logic based one.

If we observe the module declaration given in Figure 4.4, we can see that deductive knowledge is a basic component of modules as seen in Chapter 2. The components of the deductive knowledge are the dictionary, the rules and the inference system declarations.

```
Module foo =
    ...
    Deductive knowledge
        Dictionary: ...
        Rules: ...
        Inference system: ...
    end deductive
    ...
End
```

Figure 4.4: Deductive declaration into the modules.

Now we relate the components of the deductive knowledge of a module with those of the primitive syntax described in Section 4.2.1. A dictionary declaration contains fact declarations, that is the set of concepts that will be used into a module (*temperature*, *feber*). Facts can be declared to belong to a type. For instance we can say that *temperature* is a numeric fact and *feber* is a logic fact. Then, the value of the first fact will be a real number and the value of the second one an interval of truth–values. The atomic formulas of this language (atoms in the primitive syntax) are facts of logic type or predicates over facts of another type. For instance an atomic formula with the fact *temperature* could be *temperature* $> 36.5°$. Rules are composed by this kind of atomic formulas. Then, the sentences of the deductive knowledge of **Milord II** are composed by pairs of facts and their value (different from the mv–atoms) and rules weighted by intervals of truth–values of the form $[a, 1]$ (equal to mv–rules, but with different atomic formulas). All this is explained in detail in the following paragraph.

The inference system declaration was explained in Section 3.4. Remember that it declares the local logic of a module. The set of linguistic terms $A_n$ is declared as a part of an inference system declaration.

Here we will present the concrete syntax of the dictionary and the rules of deductive knowledge that allows us to write real applications. Syntax is introduced mainly to introduce the different constructs. We do not describe here

all the possible syntactical forms that are valid in the language. Please reference the complete syntax in Appendix A. When possible we use real examples to illustrate the programming task with **Milord II**.

### 4.4.1 Facts

Facts are one of the most primitive components of the language **Milord II**. They are atomic symbols that represent the concepts that will be used in a module. Facts can be observable (fever), deducible (pneumonia) and so on.

The declaration of a fact is composed by an atomic name that is the identification of the fact and a set of attributes: name, question, type, function and relations. Facts can be referenced inside a module[9], for instance in its rules.

Facts are declared in dictionary declarations. An example of dictionary is given in Figure 4.5. It contains type and fact (predicate) declarations. In this example we can see the declaration of a fact identified by *assoc_treat*.

**Dictionary:**
    **Types:** farmac = (carbamacepina, teofilina, digoxina,
        dicumarinics, ciclosporina,
        difenilhidantoina)
    **Predicates:**
        assoc_treat=
            **Name:** "Associated Treatments"
            **Question:** "which farmacs the patient takes usually?"
            **Type:** farmac
            **Relation:** needs_true use_farmacs?

Figure 4.5: Example of dictionary declaration

#### Attributes of Facts

Here we will describe all the attributes that facts have. Name and type attributes of facts are mandatory.

**Name:** This is an attribute that associates a long name to the fact. Experts use atomic short names to identify a fact mainly to simplify the rules and metarules. The long name is presented to the user for helping its understanding of the concept the fact represents.

**Question:** This is the text of the question made to the user when a fact belongs to the import interface. When the system asks a question to the user it presents the long name of the fact, the question and the set of possible

[9]A module can reference all the facts declared into the module. Facts declared into other modules can be referenced by using a path name if they are accessible (remember that they can be hidden by the information hiding mechanism).

answers among which the user can select. When a fact belongs to the import interface of a module this attribute is mandatory.

**Type:** It declares the type of the fact. The type of a fact is the set of values that it can take when it is evaluated. There are five predefined types, that is, boolean, logic, numeric, array and class; and one user–defined type named enumerated. A fact can be evaluated over the set of values determined by its type.

**Function:** Sometimes we want to evaluate a fact with a procedural form instead of a deductive one. This allows us to evaluate a fact by means of a functional description.

**Relation:** It stablishes named relations with another facts. This implements a directed graph of relations among facts that can be used to classify facts, to stablish an order of evaluation, etc.

We can see the example in Figure 4.5. This fact has a name, a question because its value will be obtained from the user, and the type (declared in the type declaration of the dictionary) which is a set of farmacs. The relation declaration means that the fact *use_farmacs?* needs to be known true to give sense to the fact *assoc_treat*.

After this summary description of the attributes of facts, we will explain in detail the type, function and relations attributes.

### Types of facts

In the first part of this Chapter we have worked with facts weighted by an intervals of truthvalues. Now we say that this kind of facts are of logic type. We need to add more types that the logic one. We follow *Milord* in this but with some variations. There are five predefined types: boolean, logic, numeric, array and class. The user can define new types of facts by enumeration, then we say that facts are of order $0^+$. Here there is a summary of the types of **Milord II** facts:

**Boolean:** Are facts whose value can be *Yes* (*true*) or *No* (*false*). They are used when we want talk about the presence or absence of a concept. For instance we can consider that the fact *has_fever* is a boolean fact because we have a criteria to decide if it is true or not the patient has fever.

**Logic:** The values of this facts are intervals[10] of a set of linguistic terms that represents uncertainty values. This set of terms must be defined in the inference system declaration (see Section 3.4). The kind of concepts that can be considered logic facts are those whose truthness can be valued, because there are subjective. For instance if we use a subjective criteria to

[10] *Milord* used only one element of a set of linguistic terms. This type of facts was named *fuzzy*.

appreciate if a patient has fever as touching with the hand, we can consider that the fact *has_fever* is a logic fact (we can say that *has_fever* is *possible*).

**Numeric:** The value is an interval of real numbers. They are used in quantitative data, for instance temperature, number of leucocits, etc.

**Enumerated:** This is the only type that can be defined by the user. This type is a set of symbolic values. The value of a fact of enumerated type is a subset of the type declaration, or *none*. A difference with the enumerated type in *Milord* is that each symbol of a enumerated fact has a certainty value associated (it is a fuzzy set as shown in Section 3.2.3). In Figure 4.5 we can see an example of enumerated type declaration *farmac*. An enumerated type can be defined in the type attribute of a fact declaration. It can also be defined in the type declaration of the dictionary (see Figure 4.5) and give it a name. This name can be used to declare a type of a fact.

**Array:** The value of this kind of facts are arrays of real numbers of any dimension. This is a special type of fact and it is only used in an example of *belief propagation* application described in the Section 6.6.

**Class:** The facts of this type have no value. They are used only to define relations with other facts. For instance we can consider that the fact *oral* is a class fact because we want to define a relation with it in the declaration of the antibiotics that are administrated orally.

The facts that belong to the import interface of a module can be asked to the user[11]. He can answer with the set of values that are allowed depending of the type of the fact. Users are able to answer *unknown* to a system question. It is very important to distinguish this answer of the interval $[0, 1]$. In the first case the system considers that the fact questioned provisionally has no value. Then, it do not produce any actions on the module, it is not used to specialize the KB. The interval $[0, 1]$ is an interval of truth–values used internally that represents the definitive value *unknown*.

### Fact Functions

Fact functions allows us to implement a set of functionalities that are useful in the practice of ESs. Here we emphasize possible applications of functions as interfaces with other programs, procedural evaluations of facts and fuzzy sets.

The function attribute of a fact is programmed in Common Lisp[12]. It is a *lambda–expression* without parameters. This *lambda–expression* is considered to be into a *lexical closure* that contains the name of the facts declared into a module. The evaluation of these fact names returns their value. Then the *body*

[11] Facts of the import interface are not always asked to the user. In some cases the control of the module can give a value to an imported fact before asking it to the user.

[12] Underline language of **Milord II** is Common Lisp, then it is easy to introduce part of the ES programmed in this language. In this Section we will use Common Lisp terminology that is considered to be well known. Details of this can be found in (Steele, 1984).

of the *lambda–expression* can contain fact names and they will be substituted by their values.

The value of a fact that contains a function attribute in its declaration is calculated by means of the evaluation of that function. To evaluate this kind of fact its function attribute is evaluated and the result of this evaluation is attached to the fact. Now we can see a set of examples of the function attribute.

One of the characteristics of **Milord II** commented in the introduction is that it should work in a realistic computer environment. Function attributes can provide a tool for communicating an ES with other programs. For instance in *Spong–IA* application the system asks the user which is the form of the sponge he wants to classify. Then it is more clear if the question is joined with a set of pictures representing the possible forms. The user can select one of them by means of a picture representation program. For instance, the evaluation of the fact *form* can be obtained by means of a function that uses that program. Function attributes can be used as a sort of interface with other programs as graphics, databases, etc.

Sometimes the value of a fact can be obtained easily by means of a mathematical expression or a procedure. An example of function attribute of *Terap–IA* application can be seen in Figure 4.6. The value of the fact *clearance_of_creatinin* can be obtained by means of the following expression. In Figure 4.6 we can see the declaration of this fact *creat_clear* that contains a function attribute.

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

```
Creat_clear=
      Name: "Clearance of creatinin"
      Type: numeric
      Function:
            #'(lambda ( )
                  (- 140
                  (/
                        (* age weight (case sex
                                    (male 1.0)
                                    (female 0.8)))
                        72)))
```

Figure 4.6: Example of function attribute

Another use of the function attribute of a fact is for declaring the characteristic function of a fuzzy set (see Section 3.2.3). In Figure 4.7 we can see an example of fact declaration of the concept *tall*. This concept is represented by a fuzzy set and declared with a function that given an interval of numeric values

that represents the height of a person returns an interval of truthvalues. You can see a graphical representation of this characteristic function in Figure 3.2.

```
Tall=
    Name: "Tall"
    Type: logic
    Function:
     #'(lambda ()
        (labels ((tall (omega)
            (cond
                ((< omega 1.7) 'impossible)
                ((and (>= omega 1.7)
                    (< omega 1.75)) 'sli_possible)
                ((and (>= omega 1.75)
                    (< omega 1.8)) 'possible)
                ((and (>= omega 1.8)
                    (< omega 1.9)) 'very_possible)
                ((>= omega 1.9) 'definite))))
        (list (tall (first height)) (tall (second height)))))
```

Figure 4.7: Example of characteristic function.

In the Section 6.5 we can see an application that uses the function attribute to implement fuzzy sets.

### Fact Relations

A fact can have several relation attributes. Relation attributes define named relations between the fact and other facts of the system. We can distinguish between relations defined by the user, and predefined relations. The expert can define relations among facts and give them sense and properties in the control component of the module. Relations can be used in the conditions of metarules, this will be explained in Section 5.4.1. Here we present some examples of relations defined by the expert of *Terap–IA* application:

- *eritro_DB equivalent_spectrum doxi* : The facts *eritro_DB* and *doxi* are antibiotics. This relation means that both antibiotics can treat the same kind of pneumoniae.

- *amoxi belongs_to_group administracio_oral* : The antibiotic *amoxi* belongs to the group of the antibiotics administrate orally.

- *vanco_tract agree_with anam_spec/insuf_renal* : Notice that we can define relations with visible facts of other modules, in this case the relation of fact *vanco_tract* with the fact *insuf_renal* of the module *anam_spec*.

There are a set of common problems faced by experts that can be solved using relations. Predefined relations are a set of relations with a global meaning in the system.

**Belongs_to:** Experts usually use a belonging relation in their applications. This is a predefinited relation to avoid defining its transitive property, giving a more efficient system. For instance when the expert defines these three relations among antibiotics, *doxi belongs_to ABS/tetras_2*, *doxi_DI belongs_to ABS/tetras_2* and *tetras_2 belongs_to tetraciclines*, the system adds the new relations *doxi belongs_to tetraciclines* and *doxi_DI belongs_to tetraciclines*.

The relations of type needs are used to give a correct ordering of questions to the user in the case of importable facts.

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

**Needs_true:** This case is similar to the last one but the behavior is different depending on the answer of the first question. Consider the following example of relation from *Spong–IA*, *organization needs_true foreign*. The first question to the user would be if the sponge is *foreign*. If the user answers "yes", the question *organization* would be asked. If the user answers "no", then the fact *organization* would become *false* and no question about it would be made.

**Needs_false:** The only difference with the last case is that the behavior is the inverse one with respect to the answer to the first question.

The needs relation work with importable facts of type boolean logic or enumerated. In the case of enumerated facts we consider that *false* is the *none* answer and that *true* is any answer but *none*.

If the expression has the value *unknown*, then the fact takes the value *unknown*. If the expression is evaluated with false in the case of boolean and logic facts, and with none in the case of enumerated facts, then the fact takes the same value depending on the type of the fact.

Facts can be evaluated by the user (importable facts) , by the function attribute, or deduced by the rules. Before the above evaluation of a fact all the *needs* relations are solved.

### 4.4.2 Rules

Syntax of **Milord II** rules is given in Figure 4.8. It is not necessary to give a complete explanation of rules because they are similar to the mv–rules described before. The only difference is that the conditions of rules can be composed of predicates on the facts of different type (conditions in mv–rules were literals of type logic).

| | | |
|---|---|---|
| rules | ::= | rule rules \| rule |
| rule | ::= | ruleid **<u>If</u>** premisse-rule **<u>Then</u>** conclusion-rule [*documentation*] |
| premisse-rule | ::= | condition-rule **<u>and</u>** premisse-rule \| condition-rule |
| conclusion-rule | ::= | **<u>conclude</u>** rconclusion **<u>is</u>** certainty-value |

Figure 4.8: Syntax of the rules.

The rules are composed of an identifier, the premise (that is, a conjunction of conditions), the conclusion, and the certainty value of the rule. The certainty value of a rule is a linguistic term belonging to the set $A_n$ of the local logic of the current module. Internally, this linguistic term is translated to an upper–interval $[a, 1]$.

### 4.4.3 Predicates on Facts

Facts are used to build the rules. They appear in the conditions and the conclusion of rules. The evaluation of a condition or a conclusion is always an interval of truthvalues. Class facts have no value and they do not appear in the rules. Then in the case of facts where their evaluation is not an interval of truthvalues we must predicate on them. That is the case of numeric and enumerated facts.

First of all we explain the conditions of rules. We can see in Figure 4.9 the syntax of the conditions of rules. In order to give an understandable explanation to conditions of rules, we do not consider conditions containing paths.

The conditions of a rule can be written in affirmative form (*condition*) or in negative form (*no(condition)*). The evaluation of *condition* returns an interval of truthvalues, then all the syntactic categories of type *condition* can be negated using the negation operator $N_n^*$ of the logic of the current module.

The elemental conditions can be:

1. A fact name or a certainty value belonging to the local logic of the module, including *true* or *false*.

2. A formula composed of facts of any type.

3. Operations among expressions containing facts of type numeric and enumerated.

A condition can be a certainty value. This is the most simple case because certainty values are self–evaluated symbols. They can be used to give initial values to facts, for instance:

**If** definite **then conclude** ciprofloxacine **is** *definite*

condition-rule ::= condition | **no** (condition )<br>
condition ::= certainty-value |<br>
pathform |<br>
operator (expression, expression )<br>
pathform ::= pathform-s | pathform-c<br>
pathform-s ::= *predid* | *amodid*/pathform-s<br>
pathform-c ::= (formula )| *amodid*/pathform-c<br>
formula ::= (pathform op pathform )<br>
operator ::= ≤ | ≥ | <= | >= | ≡ | /= | **int**<br>
expression ::= number |<br>
operator-arit (expression, expression )|<br>
operator-set (expression )|<br>
pathform-s |<br>
(values )<br>
values ::= values-crisp | values-fuzzy<br>
values-crisp ::= *symbol* , values | *symbol*<br>
values-fuzzy ::= (*symbol* , *symbol* ), values-fuzzy |<br>
(*symbol* , *symbol* )<br>
operator-arit ::= + | - | * | :<br>
operator-set ::= **cut** | **core** | **support** | **complement**<br>
op ::= **plus**

Figure 4.9: Syntax of the conditions of rules.

This kind of rule can be fired immediately and the value of *ciprofloxacine* will be *definite*. A condition can be a fact name when the type of this fact is boolean or logic. Evaluation of logic facts returns and interval of truthvalues. Boolean fact evaluation returns a boolean certainty value (*true*, *false*). Then the value *true* is assimilated to the last linguistic term of $A_n$ ($a_n$), and the value *false* to the first linguistic term of the chain $A_n$ ($a_0$).

### Formulas

Sometimes it is necessary to work with combinations of facts. Formulas are operations among facts of any type. The value of a formula can only be defined by firing a previous rule, and it does not have any relation with the individual values of the facts that compose that formula. For instance consider the following rule:

R024 **If** AD/pregnant **then conclude** (macrol **plus** RFM) **is** *possible*

The module that contains this rule must have the declarations of the facts *macrol* and *RFM* in the dictionary declaration, then the rule *R024* can give a value to the formula *(macrol plus RFM)*, and then this formula can appear in the condition of other rules. The expert can use any operator by previous definition of its algebra[13] or use the predefined operations.

Practical use of the predefined operation *plus* is given in Section 6.2.3, *Terap–IA* use this kind of operation to produce a pneumonia treatment composed of antibiotic combinations. This is an algebra with the following properties:

Symmetric: (a plus b) = (b plus a)
Associative: (a plus (b plus c)) = ((a plus b) plus c)

A module can export a formula when all the components of that formula are exportable facts.

Before explaining the operations between expression we should distinguish between operations composed by numeric facts and those composed by enumerated facts.

### Numeric Operations

A numeric expression is composed by numbers, numeric facts and arithmetic operations (+,-,*,:) among them. The evaluation of a expression of this type returns a number. Then we can apply the relations of Table 4.3 to numeric values.

The valid relations are < (less), > (greater), <= (less or equal), >= (greater or equal), = (equal), and / = (different). These are overloaded operations in the sense that these operations are different depending on the type of the expressions that are involved. Now we can see the operations with enumerated facts.

[13]Now the language **Milord II** has not the constructs to define those algebras. Only the predefined operator *plus* can be used.

### Enumerated Operations

Remember the example in Figure 4.5. The enumerated fact is a fuzzy set named *assoc_treat*. When an enumerated fact is evaluated we consider the reference set to be the type attribute of the fact, in this case a set of antibiotics:

The operations "+" and "-" are interpreted as the set union and the set intersection. This operators are applied to fuzzy sets as shown in Section 3.2.3.

The valid relations are $<$ (subset), $>$ (superset), $<=$ (subset or equal), $>=$ (superset or equal), $=$ (equal), $/=$ (different) and *int* (intersection). The operators apply over the evaluations of the expressions. These operators only apply between expressions of the same type, and the result is an interval of truthvalues. We explain the sense of the operations by means of the Table 4.3.

| a / b | **numeric** | **enumerated** |
|---|---|---|
| $<$ | $a < b$ | $R_{\subset}(A, B)$ |
| $>$ | $a > b$ | $R_{\supset}(A, B)$ |
| $<=$ | $a \leq b$ | $R_{\subseteq}(A, B)$ |
| $>=$ | $a \geq b$ | $R_{\supseteq}(A, B)$ |
| $=$ | $a = b$ | $R_{=}(A, B)$ |
| $/=$ | $a \neq b$ | $R_{\neq}(A, B)$ |
| *int* | no sense | $R_{\cap}(A, B)$ |

Table 4.3: Operations between expressions.

The syntax of the conclusion of rules is given in Figure 4.10. They are simpler than conditions. Notice that in the conclusion of rules they can not appear paths because the module only can conclude local facts.

| | | |
|---|---|---|
| rconclusion | ::= | form \| |
| | | ( *predid* ≡ values )\| |
| | | **no** ( *predid* )\| |
| | | **no** (form *op* form )\| |
| | | **no** ( *predid* ≡ values ) |
| form | ::= | *predid* \| |
| | | (form *op* form ) |
| values | ::= | *symbol* **or** values \| *symbol* |

Figure 4.10: Syntax of the conclusion of rules.

Conclusions can be of affirmative or of negative form. They can conclude facts or formulas.

## 4.5 Conclusions

We have presented the deductive knowledge of the modules of **Milord II**. It is composed of weighted facts and rules and it has a set of added functionalities that contribute to the real needs of ESs.

The inference engine of **Milord II** is based on specialization of KBs. This allows us to have an enriched behavior of the ES, consisting in the improvement of the communication with the user, the results provided to the users and the validation process.

# Chapter 5

# Control

In the precedent Chapters we have explained the modular architecture, the approximate reasoning representation and the deductive mechanisms of **Milord II** based on specialization. We can program an application by defining a hierarchy of modules, their interfaces and their deductive knowledge composed by facts and rules. Furthermore we declare the concrete local logics of every module and the translation mechanisms between these logics. The next step is to explain the control.

Till now we have given an approximate idea of the operational semantics of **Milord II**. It can be summarized by the following statements:

1. The user queries a visible module for the value of an exportable fact.
2. The module obtains external data from the user and it also makes queries to its submodules[1]. The answers given by the submodules are then translated to the actual local logic if necessary.
3. With this new information the module specializes its deductive knowledge.
4. Steps 2 and 3 are performed by order to find a solution to the initial query.

This is a global view of the operational semantics of **Milord II**, but we should make precise the details of the whole execution. For instance, given a query to a module, the system must decide which are the facts that will be necessary to give a solution to that query. Which facts will be asked to the user and which will be queried to its submodules. How to use the set of rules of the deductive knowledge of the module to deduce a fact, in which order and how.

In this Chapter we will clarify the execution of an ES programmed with **Milord II**. For that we start by introducing the main ideas related to control.

**Local Control:** We have decided to introduce the control locally to every module. This allows us to identify a module as the complete description of

[1]Returning to the first step for this submodule.

a problem. The separation between domain and control knowledge is a mandatory characteristic of ES's languages to provide a clear and declarative programming style.

**Specific versus general knowledge:** Experts usually have different methods to reason on a problem depending on the amount of data they know. For instance, a physician do not have aspirations to know all the data about the patient to make a diagnosis. If the patient is in a coma he can not ask questions to him, but he should make a diagnosis despite the lack of these data. To represent these situations experts program rules with different levels of specificity (using more or less information from the patient, that is, putting more or less conditions) to deduce the same fact. These kind of rules allows us to deduce a fact using more specific or more general knowledge. **Milord II** extends the concept of subsumption of *Milord* by using *partial labels* in the rules. With this technique we try to use the more specific knowledge when possible.

**Avoiding unnecessary work:** It is normal to have different ways to find a solution for a problem. Furthermore these ways can have different levels of credibility. It is very important to know which is the more satisfactory way to start the reasoning, how to change it in the case of failure, and how to follow the reasoning. For instance a physician can dispose of different laboratory analysis to find a germ. He starts first with the better analysis but if it fails he chooses another. **Milord II** uses specialization in order to detect if a rule can not yet improve the result of the current goal. During the execution of a case, the deductive knowledge of modules is specialized with the new known facts. Specialization produces new rules changing the premises and the uncertainty values of the previous rules. When a rule is fired, we can decide if the other rules that conclude over the same fact are able to improve the result obtained with the first one. If they can not we should eliminate those rules so avoiding unnecessary work.

**Precision Level Results:** We have seen that the certainty value of facts is represented by an interval of truth–values. These intervals represent more or less precise values of facts. In some cases experts are interested in programming modules whose results should be greater than a minimum precision level. They are not interested in less precise results. Experts can declare a *threshold* to fix when the values of the facts of a module are significant, that is, when they can be used to produce precise results. For instance, we can program a module that only considers that a fact is significant when its certainty value is greater than *possible*.

**How to obtain Data:** Given a query to a module we can follow different strategies to answer the query. How to obtain the external facts of the module and in which order are the essential points of the different evaluation strategies. **Milord II** allows to declare different evaluation strategies because of the separation between the search process and the deductive one

(specialization).

**Programming Control:** **Milord II** provides Horn–like metarules that can be programmed by experts. It is a powerful method of local control that can simplify the deduction by eliminating rules, deducing facts and changing the hierarchy of modules.

We divide this Chapter between the implicit control, that is, the preprogrammed characteristics of the execution; and the explicit control as the parameters of control that the expert can declare.

Implicit components of control are build–in in the interpreter and then they can not be programmed. Implicit control is composed of two mechanisms, the unnecessary rules detection and the subsumption treatment. They act when we have more than one rule that conclude over the same fact.

```
Control knowledge
        Evaluation Type:  ...
        Truth Threshold:  ...
        Deductive Control:  ...
        Structural Control:  ...
end control
```

Figure 5.1: Control declaration

Explicit components of control, that is, those that can be programmed by experts are explained after implicit ones. These components of control knowledge are the threshold, the evaluation strategy and the reification and reflection mechanism related to metarules. Figure 5.1 shows the syntactical declaration of the explicit control in a module.

## 5.1 Implicit Control

Subsumption control and unnecessary rules control are important characteristics of the execution of a module that can not be programmed by the expert.

### 5.1.1 Subsumption

We have repeated along this thesis that the kind of knowledge managed by ESs is imperfect, that is, incomplete, imprecise and uncertain. Incompleteness is an usual characteristic of the knowledge managed by human experts. They have an special ability to manage and obtain useful conclusions from situations with more or less complete knowledge. For instance, a physician needs to know data of the patient to decide an adequate treatment. Nevertheless if the patient is in a coma then the physician should find a treatment using less data. An ES

should be capable to modelize this kind of behavior allowing to program different solutions to a problem with different levels of incompleteness. When possible the ES should work with the less incomplete (more specific or more specialized) knowledge as an expert does. That is the criteria used in **Milord II**.

Experts express the deductive knowledge of a module by mean of rules. Frequently they write several rules containing the same fact in its conclusions. These rules can represent disjunctive paths in the proof tree of that fact, but in other cases they can represent the same path with different incompleteness level[2]. We will show two simple examples to clarify this concept. Consider the first example from *Terap–IA*:

R051 **If** cotri **then conclude** cotri_DB **is** slightly_possible
R052 **If** cotri **and** seriousness **then conclude no**(cotri_DB) **is** definite

These two rules conclude over the same fact *cotriDB*. It is easy to see that rule *R052* is more specific than rule *R051*. Whenever we can apply the more specific rule, we could also apply the more general one. We say that there is a subsumption relation between these rules. This is an example of lack of data. In this example the expert has a default rule for the case where they have no sufficient evidence[3] of the seriousness of the illness. Then the fact *cotri_DB* can be deduced with less data.

If we consider that the fact *cotri* is *true* and the fact *seriousness* is *unknown* the value for *cotriDB* would be $[slightly_possible, 1]$ using the more general rule. If we know that the fact *seriousness* is *true* we will use the more specific rule, and the result for the fact *cotriDB* would be $[0, 0]$[4] or *false*. Notice that it is not necessary that the truth–value of the more specific rule to be greater than the general one. The second example is:

R001 **If** age $> 60$ **then conclude** old **is** possible
R002 **If** age $> 70$ **then conclude** old **is** very_possible

Intuitively it is clear that the rule *R002* is more specific that the rule *R001*. It is more specific to know that the age is greater that seventy than the age is greater that sixty.

In **Milord II** as in *Milord* we first apply the most specific rules. If the most specific rule can not be applied, then the most general one is fired.

In this sense we can say that in some cases the more specific rule can not be applied but only the more general can (default reasoning). In the cases where the more specific rule can be applied, the more general could be applied also.

[2]In Chapter 4 we had only considered that the rules with the same fact in the conclusion were disjunctive paths.

[3]We say sufficient evidence in the sense that the rule would conclude a value different than unknown ([0,1]). We will insist in this aspect in the next Section where we will talk on the threshold control component.

[4]Notice that without the subsumption criteria these two rules would produce an inconsistent result when both would be applied ($[slightly_possible, 1] \cap [0, 0] = \emptyset$).

Our criterion is that we always apply the more specific rule if possible and in that case the more general rule will not apply.

Remember the first example. If we know the fact *seriousness* then we only use the rule *R052*. In the other case we will use rule *R051*.

Now we analyze the above subsumption criterion from a general point of view in the concrete syntax of **Milord II** rules.

### General Subsumption

Here we explain the general subsumption criterion. We think that it is interesting to use always the more specific knowledge in the deductive process, but we will finally define a criterion that is a compromise among complexity and understandability.

First of all we analyze the subsumption criterion between two isolated rules.

**Definition 5.1 (Subsumption)** *Given two rules $R_1$ and $R_2$ with premises $A$ and $B$ respectively, and that they conclude over the same fact, we say that rule $R_1$ is more specific than rule $R_2$ or that rule $R_2$ is more general than rule $R_1$, when*

$$(A \to B, 1)$$

Now we can simplify this criterion taking into account the different types of premises of **Milord II**.

First of all we can simplify the criterion using a property of the implication (**I1**, Section 3.1) used in **Milord II**.

$$(A \to B, 1) \Rightarrow \rho(A \to B) = 1 \Rightarrow I_T(\rho(A), \rho(B)) = 1 \Leftrightarrow \rho(A) \leq \rho(B)$$

Premises of rules are composed of a conjunction of predicates over facts. Facts can be of four types: boolean, logic, numeric and enumerated. We can only compare facts of the same type. We extend the subsumption criterion taking into account the different components of the premises of the rules.

It is easy to see from properties **T5** and **T1** of $T$ functions that the following holds:

If $a \leq b$ and $c \leq d$ then $T(a, c) \leq T(b, d)$

Proof:

$$\left.\begin{array}{l} a \leq b \Rightarrow T(a, x) \leq T(b, x), \forall x \\ c \leq d \Rightarrow T(c, y) \leq T(d, y), \forall y \end{array}\right\} \Rightarrow T(a, c) \leq T(b, d)$$

We can consider that premises of rules are composed of a conjunction of sets of predicates over facts grouped by their type. $B$, $L$, $N$, and $E$ are sets of predicates of types boolean, logic, numeric and enumerated respectively. Then it is easy to extend the last property to the new one:

$$\left.\begin{array}{l} \rho(B_1) \leq \rho(B_2) \\ \rho(L_1) \leq \rho(L_2) \\ \rho(N_1) \leq \rho(N_2) \\ \rho(E_1) \leq \rho(E_2) \end{array}\right\} \Rightarrow \rho(B_1 \wedge L_1 \wedge N_1 \wedge E_1) \leq \rho(B_2 \wedge L_2 \wedge N_2 \wedge E_2)$$

This allows us to group the conditions on sets of the same type and to apply the subsumption criterion to each group separately. Now we should test if the following set of conditions hold:

$$\{(B_1 \rightarrow B_2, 1), (L_1 \rightarrow L_2, 1), (N_1 \rightarrow N_2, 1), (E_1 \rightarrow E_2, 1)\}$$

### Boolean and Logic premises

Suppose the following rules:

BL1: $(a_1 \wedge a_2 \wedge \cdots \wedge a_n \rightarrow c, \alpha)$
BL2: $(b_1 \wedge b_2 \wedge \cdots \wedge b_m \rightarrow c, \beta)$

If we want to test if the rule *BL1* subsume the rule *BL2* then:

$$\rho(b_1 \wedge b_2 \wedge \cdots \wedge b_m) \leq \rho(a_1 \wedge a_2 \wedge \cdots \wedge a_n)$$

If $m \geq n$ and we can found $n$ facts such that $a_i = b_j$ then it is easy to see that $\rho(a_1 \wedge \cdots \wedge a_n) = \rho(b_1 \wedge \cdots \wedge b_n)$. And by the property **T5** of $T$ functions the following expression always hold.

$$T(\rho(a_1 \wedge a_2 \wedge \cdots \wedge a_n), \rho(a_{n+1} \wedge \cdots \wedge a_m)) \leq \rho(b_1 \wedge b_2 \wedge \cdots \wedge b_n)$$

In this case it is easy to see that the subsumption criterion is that rule *R1* is more general than rule *R2* when $A \subseteq B$, where $A$ and $B$ are the set of boolean and logic conditions of rules *BL1* and *BL2* respectively.

### Numeric and enumerated premises

This case is not so easy as the logic and boolean premises. Numeric expressions can be a set of arithmetic expressions among facts and numbers. Each condition can contain several facts, and two conditions can predicate over the same fact. Consider the following example:

$(a - b > 3 \wedge b < 6 \rightarrow c, \alpha)$
$(a < 12 \rightarrow c, \beta)$

In this case it is easy to see that the second rule subsumes the first one, because:

$$\rho(a - b > 3 \wedge b < 6) = \rho(a > b + 3 \wedge b < 6) = \rho(a < 9 \wedge b < 6) = T(\rho(a < 9), \rho(b < 6)) \leq \rho(a < 9) \leq \rho(a < 12)$$

In general the problem can be reduced to two inequation systems, each one composed by the conditions of each rule. After finding the solutions of the two systems, we can compare each variable of the systems, and show if one system implies the other. In the last example:

$$\left.\begin{array}{l} a - b > 3 \\ b < 6 \end{array}\right\} \Rightarrow \left.\begin{array}{l} a < 9 \\ b < 6 \end{array}\right\} \Rightarrow a < 12$$

Finding solutions for an inequation system is a complex task. It is easy to see that in the case of enumerated premises the problem is similar, but much more complicated (remember the operations on fuzzy sets explained in the Section 3.2.3).

In real programs experts normally do not write very complicated conditions. Furthermore a complex subsumption criterion is an impediment more than a help to the understandability of subsumption when the expert writes his program.

For all these reasons in the case of numeric and enumerated premises we consider that there is a subsumption relation between two rules when the facts contained in the premise of one rule are contained in the set of facts of the premise of another rule as in the logic and boolean case.

### Subsumption in depth: Partial Labels

There are added problems that have not been addressed yet. Till now we have only worked with two isolated rules. The last criterion is not enough for catching all the subsumption relations when we have many rules. For instance consider the following set of rules:

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

It is easy to see that there is a hidden subsumption relation between the rules $R_1$ and $R_2$. Considering the usual chaining of rules we can obtain that the set of facts necessary to fire the rule $R_1$ is $\{a, b, c, d\}$, and for the rule $R_2$ is $\{a, b, c\}$. $R_1$ is more specific than $R_2$. In this case we do not only compare the premises of the rules but the set of facts in the deduction tree below the fact along two different paths.

We should not forget that we are thinking on an isolated module. If we consider all the modular structure then we can find new hidden subsumption relations. Suppose that the rules in the last example are distributed in two different modules.

$$M_1 : \begin{cases} R_1^{M_1} : M_2/f \wedge M_2/c \wedge d \rightarrow g \\ R_2^{M_1} : M_2/e \wedge M_2/f \rightarrow g \end{cases} \quad M_2 : \begin{cases} R_1^{M_2} : c \rightarrow e \\ R_2^{M_2} : a \wedge b \rightarrow f \end{cases}$$

We can find the same subsumption relation than in the first example.

In our system we only find local subsumption relations. Other subsumption relations are ignored.

Now we describe a partial solution to this type of hidden subsumption relations using *partial labels*.

The first idea is to find the set of facts in the deduction tree below the facts. We would compare them in order to find the subsumption relations. Apart of the complexity of this task, there is another problem. We consider subsumption with respect to the data that has been really used to deduce a fact. We do not

know *a priori* the data that will be necessary to do this. We can not predict which of the conjunctive paths will be successful. Then we can not build labels statically but at runtime.

Now we explain the method used in **Milord II**. This is based on *partial labels*. Specialization allows us an easy implementation of this kind of task.

We will illustrate the explanation with the example in the Figure 5.2. The complete label of a fact in a rule is composed of terminal nodes of the proof tree considering an isolated module. Terminal nodes of a module are the imported facts and the *prefixed* facts (those facts belonging to the submodules).

$$\left.\begin{array}{l} R_1\langle a,b,c,d\rangle : a \wedge b \wedge c \wedge d \rightarrow g \\ R_2\langle\rangle : e \wedge f \rightarrow g \\ R_3\langle c\rangle : c \rightarrow e \\ R_4\langle a,b\rangle : a \wedge b \rightarrow f \end{array}\right\} \stackrel{a}{\leadsto} \left.\begin{array}{l} R_1\langle a,b,c,d\rangle : b \wedge c \wedge d \rightarrow g \\ R_2\langle\rangle : e \wedge f \rightarrow g \\ R_3\langle c\rangle : c \rightarrow e \\ R_4\langle a,b\rangle : b \rightarrow f \end{array}\right\} \stackrel{b}{\leadsto}$$

$$\left.\begin{array}{l} R_1\langle a,b,c,d\rangle : c \wedge d \rightarrow g \\ R_2\langle a,b\rangle : e \rightarrow g \\ R_3\langle c\rangle : c \rightarrow e \end{array}\right\} \stackrel{c}{\leadsto} \left.\begin{array}{l} R_1\langle a,b,c,d\rangle : d \rightarrow g \\ R_2\langle a,b,c\rangle : \emptyset \rightarrow g \end{array}\right\}$$

Figure 5.2: Example of subsumption.

Initially we can build the partial labels of each rule identifying the facts in the premise of the rule that are terminal nodes. In our example all the conditions in the rule $R_1$ are terminal nodes. Conditions of the rule $R_2$ are not terminal nodes, they are deducible nodes (by the rules $R_3$ and $R_4$).

When a rule is specialized with a fact that has been deduced by another rule, the first rule extends its label with the label attached to the other rule. In the example when rule $R_4$ is fired and the fact $f$ specializes the rule $R_2$, the new partial label of $R_2$ is the union of the old label (empty) and of the label of the fired rule ($\langle a,b\rangle$).

Before firing a rule we can compare its label with the labels of the other rules that conclude over the same fact. In the case a subsumption relation between two rules is detected, we act as explained before. In the last specialization step of the example, we can observe that the rule $R_2$ can be fired but its label is included in the label of the rule $R_1$. In that case the rule $R_2$ is more specific than the rule $R_1$ and it is not fired until we know if rule $R_1$ is able to be fired.

Finally we can summarize the design decisions adopted for the practical implementation of subsumption relations.

- We consider only local subsumption relations inside modules. The relations that can be detected building the partial labels of the rules during execution time.

- Runtime labeling of rules allow us to detect cases of hidden subsumption relations.

- We consider that the specificity of a rule with respect to the other rules concluding the same fact is determined by the inclusion relationship of their partial labels. A rule is more general than another if its partial labels are included into those of the other rule.
- We only use the more specific rules if possible; otherwise we use the more general ones.

### 5.1.2 Unnecessary Rules

In the Section above we have seen that the evaluation of a fact can be carried out by one or more rules. All these rules can contribute to the evaluation of a fact taking into account that in the case of rules with a subsumption relation only a rule will be used. Now we only consider the set of rules that effectively can take part in the evaluation of a fact.

These rules will be specialized using a concrete search strategy by means of the Specialization inference rule. We can compose the results of these rules by means of the Composition inference rule, as seen in Section 4.2.3. It is easy to see that the intersection of intervals of truth–values leads to more precise or equal values. Each rule can contribute then to give more precision to the final result.

Suppose the situation where we have obtained a provisional value to a fact (a rule has been totally specialized). We can now consider if the remaining rules concluding the same fact can give more precision to the fact.

The maximum precision given to the conclusion of a rule is limited by the certainty value of the rule. Consider a rule with premise value $[a_i, a_j]$ and certainty value $[a_\rho, 1]$. The conclusion of the rule is given by

$$MP_T^*([a_i, a_j], [a_\rho, 1]) = [T(a_i, a_\rho), 1)] = [a_r, 1] \text{ ,where } a_r \leq a_\rho$$

Then it is easy to see that the value of the conclusion of the rule will be more imprecise or equal that the precision of the value of the rule. There are no differences in the case of negative conclusion because the negation does not change the precision. When the system obtains a new provisional value for a fact, we test if the rules associated to the fact are necessary or unnecessary.

If the provisional value is $[\rho_p^p, \rho_p^o]$ and the value of the rule is $[\rho_r, 1]$, if the conclusion of that rule is affirmative then the rule will be unnecessary if and only if $\rho_r \leq \rho_p^p$. If the conclusion is negative then the rule will be unnecessary if and only if $\rho_r \geq N_n(\rho_p^o)$.

We apply this test again when a rule is specialized, because specialization deals with more imprecise or equal truth–values of rules. This method allows us to save unnecessary deduction and avoids to demand information outside the module without necessity.

Specialization of rules results in new rules with new truth–values. Then we can apply this method when a rule is specialized. Notice that it is an improvement with respect to other system that do not use specialization (including *Milord*). They could only compare the rules before or after firing them. In the

last case all the questions related to a rule would be made. This can be avoided thanks to the previously explained test.

## 5.2 Threshold

Remember that the goal of the inference engine of **Milord II** is to obtain values with the maximum precision. Despite this we can obtain final values of facts with so little precision that an expert can consider not to be significant, that is, like *unknown*. **Milord II** introduces a parameter that controls the minimum precision of facts in order to consider that they are significant. This parameter is local to each module, then we can control the precision level needed to solve a concrete problem.

This mechanism of control is named the threshold ($Th$) of a module. It is a linguistic label belonging to $A_n$ and represents the minimum value a deduced fact must have to be significant. The default threshold value of **Milord II** is the second term $A_2$ of the chain $A_n$ of truth–values. MYCIN (Shortliffe, 1976) had certainty factors lying in the interval $[0, 1]$. The threshold used was of 0.2. An example of threshold declaration is:

**Truth Threshold:** possible

Now consider the following general rule:

$$(a_1 \wedge a_2 \wedge \cdots \wedge a_m \rightarrow b, [\rho_r, 1])$$

We can calculate the final value of the conclusion[5] $b$:

$$\rho(b) = MP_T^*(\rho(a_1 \wedge a_2 \wedge \cdots \wedge a_m), [\rho_r, 1])$$

$$\rho(a_1 \wedge a_2 \wedge \cdots \wedge a_m) = T^*(\rho(a_1), \rho(a_2), \ldots, \rho(a_m))$$

Considering that $\rho(a_i) = [\rho_{a_i}^p, \rho_{a_i}^o]$, then:

$$T^*(\rho(a_1), \rho(a_2), \ldots, \rho(a_m)) = [T(\rho_{a_1}^p, \rho_{a_2}^p, \ldots, \rho_{a_m}^p), T(\rho_{a_1}^o, \rho_{a_2}^o, \ldots, \rho_{a_m}^o)]$$

$$\rho(b) = MP_T^*([T(\rho_{a_1}^p, \rho_{a_2}^p, \ldots, \rho_{a_m}^p), T(\rho_{a_1}^o, \rho_{a_2}^o, \ldots, \rho_{a_m}^o)], [\rho_r, 1]) =$$

$$\rho(b) = [T(T(\rho_{a_1}^p, \rho_{a_2}^p, \ldots, \rho_{a_m}^p), \rho_r), 1] = [T(\rho_{a_1}^p, \rho_{a_2}^p, \ldots, \rho_{a_m}^p, \rho_r), 1]$$

And finally:

$$T(\rho_{a_1}^p, \rho_{a_2}^p, \ldots, \rho_{a_m}^p, \rho_r) \leq Min(\rho_{a_1}^p, \rho_{a_2}^p, \ldots, \rho_{a_m}^p, \rho_r)$$

Notice that the precision of the conclusion of a rule depends on the minimum truth–values of the rule and its conditions. Given a rule with minimum truth–value less than the threshold, it concludes the interval $[0, 1]$. The same happens when it exists one condition of the rule whose value is less than the threshold.

[5] Because we are talking on precision we will use only positive rules. Negative rules produce the same results on precision.

## 5.3 Evaluation Strategy

At this moment we know many aspects of **Milord II**. We have talked about the modular structure, the uncertainty, and the deductive component. We know how a knowledge base is specialized using the above implicit control. To specialize the deductive knowledge of a module it needs to know external data (from the user and from its submodules). But until now we have not explained how this information is obtained. In the Figure 4.2 we can see that the search process asks questions to the user and then passes information to the deductive process in order to specialize the knowledge base. In this Section we explain how the search process obtains information from the user and from its submodules.

The search process is goal directed inside each module. Given a goal, that is, a fact to be evaluated in a module, the evaluation goes by asking the necessary questions to the user and to its submodules in order to evaluate the goal with the maximum precision.

In **Milord II** there are three evaluation strategies, named lazy, eager and reified strategies. These strategies are declared locally to each module. Lazy and eager evaluation types are radical strategies in the sense that they use the minimum information in the case of lazy, or the maximum information in the case of eager. Other strategies could be defined thanks to the separation between search and deductive processes as explained in Section 4.1.1. Before explaining in detail these strategies we give a brief summary:

**Lazy:** A module with this evaluation strategy asks questions to the user and to its submodules only in the case where this questions are necessary to reach the current goal. This strategy is used by default.

**Eager:** Given a goal to an eager module the following actions are done: it asks to the user all the imported facts of the module; and it asks also all the exported facts of its submodules.

**Reified:** This kind of evaluation strategy do not differs of the eager one in the form of asking questions. We will explain in detail this kind of evaluation when explaining the reification and reflexion mechanisms in the next Section.

### 5.3.1 Lazy

A module with lazy search strategy finds the cheapest path to obtain a solution for a goal. This is a dynamic task in the sense that the module finds the next fact to be asked looking at the current state of that module (values of the facts and current rules). This determines a search cycle consisting in finding the next question that is relevant to the goal (user or submodules) and updating the module (by means of the specialization of the knowledge base). This cycle consisting on finding a question and specializing a module is repeated until the goal is reached.

We should remember which are the components of a module that participate in the evaluation of a fact. A fact of a module can be evaluated by the user (import), by its submodules, by means of *needs_true* or *needs_false* relations (see Section 4.4.1), by a function associated to a fact (see Section 4.4.1), or by the rules of the deductive knowledge of the module.

Given a goal to a module the algorithm sketched below shows how the module finds the questions that are needed to reach the current goal of the module. That process only returns which is the next fact that is necessary to obtain. This is a recursive algorithm because the initial goal to the module produces new internal subgoals that in its turn use the same algorithm.

1. *Goal of a submodule.* If the goal is a path to a submodule of the current module, and if that submodule is visible, then the algorithm returns that path (it will be asked to the submodule).

2. *Goal belonging to the import interface.* In this case we must consider if that goal has needs[6] relations with other facts. If the goal has not needs relations the algorithm returns the same goal (it will be asked to the user); otherwise the needs relations must be satisfied and then we apply recursively the function to the first fact of a non solved need relation.

3. *Goal with a function attribute.* Now the evaluation of the goal depends on the evaluation of the function attribute. In this case we apply recursively this algorithm to the first fact with no value that belongs to the function. We iterate this process until all the facts of the function are obtained. Finally the function is evaluated and its result attached to the fact.

4. *Goal that can be deduced by means of rules.* In this case we start a depth search on the rules of the deductive knowledge of the module. This will be explained in the rule search Section.

Notice that the algorithm finally returns a path to a submodule of the current module, or a fact belonging to its import interface.

### Rule Search

Before starting the rule search we order the set of rules that are able to deduce a fact with the following criteria.

1. Rules more specific first. We try to find solutions first using the more specific rules.

2. With rules with no subsumption relations we try first the more precise rules. A rule is more precise than another when its truth–value is more precise than that of the other rule. Notice that this order can change during the execution because of the specialization of rules. We try first the solutions expected more precise.

[6] Need relations are those explained in Section 4.4.1, that is, *Needs*, *Needs_true* and *Needs_false*.

3. Finally we maintain the order of rules given by the expert.

When we select a rule we maintain the writing order of the conditions of that rule (left to right). With these considerations the search strategy is depth first[7]. We apply recursively the above procedure until a fact is found.

### 5.3.2 Eager

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

The ordering of these goals is that of the declarations given by the expert, that is, first the facts of the import interface of the module with its original order, then the facts of the export interface of its submodules with the order of declaration in the submodules. In this kind of search only the *needs* relations are taken into account when asking questions to the user.

## 5.4 Reification and Reflection Mechanisms

Above we described the implicit and static mechanisms of control that are used in **Milord II**. Implicit mechanisms affect the specialization process made in the deductive knowledge of modules. We only use the rules that can improve the results (those with more precision) and those with more specific knowledge when possible. These implicit mechanisms of control can not be programmed by the expert.

The expert can declare which is the precision level of the modules and the kind of evaluation used into them. The search process is effected by the implicit mechanisms cited above and it determines which questions and in which order they will be asked to the user and to the submodules. Now we are interested in the dynamic aspects of control, that is, the control programmed by the expert by means of metarules.

We have presented a complete description of the execution of modules. Given a query to a module and depending of the kind of search strategy, the module starts making questions to the user, to other modules and making deductions by means of the specialization of modules. Now we complete the internal components of a module adding the meta control component into its structure. As introduction we explain the components shown in the Figure 5.3).

**Object Level:** The object level of a module is composed by the facts and the rules. It is an active component. Given a goal to a module, it finds that goal by inspecting the rules, relations, functions and so on, and asking questions to the user (import interface) and to its submodules (hierarchy). It follows the search strategy (evaluation type) of the module and specializes the

[7] This is not exactely true because of the *need* relations modify that search strategy.

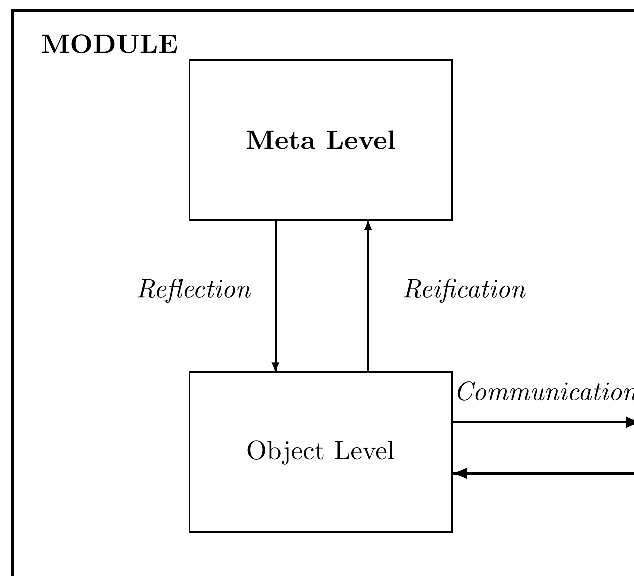


Figure 5.3: Control cycle.

rules following the implicit control (subsumption and unnecessary rules) and the specialization process with the new external information obtained. The reification process informs the meta level about the actions performed by the object level. The actions give values to facts and specialize rules.

**Meta Level:** The meta level of a module is composed of a set of metarules (classified in deductive rules and structural rules). It is a passive component. It is continuously looking at the behavior of the object level. It makes actions when the conditions of a metarule hold. The *reification* process informs then the meta level about the actions done by the object level, and the meta level acts on the object level by *reflecting* actions. These actions affects the deductive knowledge (deductive control) and the hierarchy (structural control) of the module. The actions proposed by the meta level are mandatory to the object level.

For instance, if the object level informs the meta level that the patient is a man, then the meta level reflects an action consisting in eliminating the specific rules for women.

Before describing the syntax of metarules we should see which are the components of the object level that are reified to the meta level. Static reification informs the meta level about static components of the deductive knowledge, as relations, submodules, etc. Dynamic reifications informs the meta level about the specialization process, as new deductions, specialized rules and so on.

### 5.4.1 Static Reification

The static reification informs the meta level of several characteristics of the module that do not change during the execution, as the relations among facts, the type of facts, the submodules, the threshold, the set of linguistic terms and the kind of search strategy.

**Relations:** It is the meta predicate *name_of_relation* (the predefined relations can also be used) with two arguments corresponding to the facts related (the facts can be valid paths from the module). The relation is from *fact1* to *fact2*.

name_of_relation(fact1, fact2)

**Types:** The predicate that says that a fact belongs to a concrete type:

type(fact, type_of_fact)

where $type_of_fact \in \{boolean, logic, numeric, class, list_of_symbols, name\}$. An enumerated fact is named with the set of symbols of the type of that fact (*list_of_symbols*) or its name (*name*) (see Section 4.4.1).

The static reification is performed before the execution of the module because it does not change during the execution.

### 5.4.2 Dynamic Reification

The current object level theory (OLT) is the set of rules of a module. The current meta level theory (MLT) is the set of meta–rules of a module, plus the instances of the user defined metapredicates, that were the defined in the Section above. Dynamic reification is composed of the predicates $K$, $WK$ and $P$ that are related to the value of facts, and the active submodules of the module.

First of all the minimal literal definition is given. It is composed of the set of facts deduced in the OLT with the most precise value.

**Definition 5.2 (Minimal Literal)** *We define the minimal literal OLT as*

$$OLT^* = \{(p, W) | p \text{ is a literal, and } W = \bigcap_{i=1}^{n} V_i \text{ such that } (p, V_i) \in OLT\}$$

**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:

$$\frac{(p, V) \in OLT^*}{\vdash_{\mathcal{M}} K(p, V)}$$

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

The reflection process maps the meta level theories into object level literals. The reflection rule that relates MLT with OLT is defined as:

$$\frac{\vdash_{\mathcal{M}} K(p, V)}{\vdash_{\mathcal{O}} (p, V)}$$

**Meta–predicate WK**

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

$$\frac{(p, V) \in OLT^* \text{ and } V \subseteq V^*}{\vdash_{\mathcal{M}} WK(p, V^*)}$$

$$\frac{(p, V) \in OLT^* \text{ and } V \not\subseteq V^*}{\vdash_{\mathcal{M}} \neg WK(p, V^*)}$$

**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 upwards reflection the computing of the deductive closure for $p$ is not finished neither $P(p)$ nor $\neg P(p)$ will be generated.

$$\frac{OLT \vdash_{\mathcal{O}} (p, V) \text{ and } V \neq [0,1]}{\vdash_{\mathcal{M}} P(p)}$$

$$\frac{OLT \not\vdash_{\mathcal{O}} (p, V) \text{ and } V \neq [0,1]}{\vdash_{\mathcal{M}} \neg P(p)}$$

Examples of these predicates can be:

K(fever,[very_possible,1]
WK(pneumonia,[possible,1]
P(cotri)

The first example means that the fact *fever* has been deduced in the OLT theory with value $[very_possible, 1]$ and that value is the most precise one. The second example says that the fact *pneumonia* is provisionally deduced with value $[possible, 1]$, but it could be deduced with more precision. The last example means that the fact *cotri* is proved in the OLT theory with a value different from *unknown*.

After defining which is the knowledge that the meta level contains in a concrete moment of the execution of a module, we explain the syntax of the metarules of **Milord II**. We distinguish between deductive and structural metarules. The first ones are related to the deductive knowledge of the module that contains them, and the other to the submodule structure of that module.

| | | |
|---|---|---|
| premisse-meta | ::= | condition-meta **and** premisse-meta \| |
| | | condition-meta |
| condition-meta | ::= | mconditio \| |
| | | **no(**mconditio **)**\| |
| mconditio | ::= | *metapredid* **(**conditionterm **,** ...**,** conditionterm **)** |
| conditionterm | ::= | *operation* **(**conditionterm **,**...**,** conditionterm **)**\| |
| | | *metafunctid* **(**conditionterm **,** ...**,** conditionterm **)**\| |
| | | conditio |

Figure 5.4: Syntax of the premises of metarules.

See the Figure 5.4 for the complete syntactical description of the premises of metarules. Where metapredid are the above metapredicates (*K*, *WK* and *P*) plus arithmetic predicates (*le*, *ge*, *gt* and *lt*), set operators (*member*, *diff* and *atom*). At the moment we only use *plus* as operations. Finally the metafunctions are relative to array functions (for instance *transpose*).

### 5.4.3 Deductive Control

The deductive control (see the Figure 5.5) affect the deductive knowledge of a module by inhibiting rules or deducing the above metapredicates. In some cases it is interesting to simplify the set of rules to avoid unnecessary deductions, or to deduce facts despite of the object level.

| | | |
|---|---|---|
| mrr | ::= | *metaid* **If** premisse-meta **Then** filters-mrr |
| filters-mrr | ::= | filter-mrr filters-mrr \| filter-mrr |
| filter-mrr | ::= | **inhibit rules** *relation-id* pathpredid \| |
| | | **inhibit rules** pathpredid \| |
| | | **prune** pathpredid \| |
| | | conclusion-meta |
| conclusion-meta | ::= | mconclusion \| |
| | | **no** (mconclusion ) |
| mconclusion | ::= | *metapredid* (conclusionterm , ..., conclusionterm ) |
| conclusionterm | ::= | operation (conclusionterm , ..., conclusionterm )\| |
| | | *metafunctid* (conclusionterm ,..., conclusionterm )\| |
| | | form |
| form | ::= | *predid* \| $symbol |
| | | (form op form) |

Figure 5.5: Syntax of the deductive control.

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

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

**Conclude:** Metarules can give a value to a fact of the object level. Meta level has the maximum priority and then the value of a fact given by the meta level will be definitive. This implies to inhibit the rules deducing this fact.

### 5.4.4 Structural Control

The metarules of the structural control (see the Figure 5.6) are designed to modify the hierarchy of a module by inhibiting modules or declaring new ones (dynamic modules); they can also stop definitely the execution (for instance, when the system is out of domain).

**Filter:** A metarule can inhibit (filter) submodules of a module. That means that all the facts exported by the filtered submodule will be *unknown*.

| | | |
|---|---|---|
| mre | ::= | metaid **If** premisse-meta **Then** filter-mre |
| filter-mre | ::= | **filter** amodidlist \| |
| | | **order** amodidlist \| |
| | | **Open** **(**conclusionterm **,** ...**,** conclusionterm **)**\| |
| | | **Module** **(**conclusionterm **,** ...**,** conclusionterm **)**\| |
| | | **Inherit** **(**conclusionterm **,** ...**,** conclusionterm **)** |
| amodidlist | ::= | amodid amodidlist \| amodid |
| mrx | ::= | *metaid* **If** premisse-meta **Then** exception |
| exception | ::= | **definitive solution** *predid* \| |
| | | **stop** |

Figure 5.6: Syntax of the structural control.

**Order:** When we use eager evaluation in a module, the order of questioning the submodules is by the writing order given by the expert. Sometimes it is interesting to change this order at run time. This actions allow to change this order when a set of conditions hold.

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

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

**Definitive Solution and Stop:** These are exceptional actions. In some cases the expert wants to stop the execution given the value of a fact (definitive solution) or nothing (stop). This is useful when the ES is out of domain.

Apart from the kind of process used to ask questions in the different evaluation types, we must make distinctions between them with respect to the reification and reflexion mechanisms and the specialization.

**Lazy:** Given a query to a lazy module,

1. It starts finding a question to the user or to a submodule useful to reach the current goal (see Section 5.3.1).

2. It obtains the value for this question and reifies the result (value of the fact) to the meta level.
3. The meta level tries to fire metarules. Now there is a looping between the meta level and the object level. It consists in reflecting a result to the object level (specialize, inhibit rules, filter submodules, and so on), to execute the action at the object level, and reify its results. It is an iterative process until the meta level reflects nothing.
4. Finally the object level specializes its knowledge with the answer to the first question. And it returns to the first step.

This process continues until a value for the initial query is found.

**Eager:** Given a query to an eager module,

1. It does the same actions that points 1 to 3 of the lazy strategy. The difference is that an eager module ask all the questions of its import interface and of all the export interface of its submodules.
2. Finally the module specializes with all the questions made following the same dialog with the meta level as before.

**Reified:** Given a query to a reified module,

1. The first step is similar of that of the eager strategy, but the rules of the module are also reified. In this case specialization is not used and all the deduction is made at meta level.
2. Finally the module gets the results reflected by the meta level.

Usually applications use the lazy and eager evaluations as a form of obtaining the information from the user. An example of module evaluation type *reified* is given in Section 6.6.

## 5.5 Conclusions

In this Chapter we have completed the description of **Milord II** by presenting the control. We have explained the part of the control that is implicit in **Milord II** and the one that can be programmed by the user.

# Chapter 6

# Applications

In the introduction of this thesis we put emphasis on the applicability of **Milord II** to build real world systems. After the syntactical and semantical description of all the components of the language and the system, this last Chapter deals with the systems that have been developed and that are currently running with **Milord II**.

Thanks to these application and to the enthusiastic collaboration of the experts we have been able to bring **Milord II** to the actual state. In the introduction we have distinguished **Milord II** from other systems by its purpose. The main purpose of **Milord II** are the application development. The languages directed to the applications are designed following a bottom–up methodology. The development of the applications and of **Milord II** have been in a mutual feedback cycle.

Despite the great number of problems raised by the paralell development of real applications and of **Milord II**, it has brougth a fruitful collaboration. From the developer point of view, real applications are an interesting source of new problems. Experts have taken advantage of the bottom–up development by being allowed to introduce these suggestions to the system design.

## 6.1 Introduction

We introduce the main applications developed with **Milord II**. They are different applications and each one has contributed with their own problems and solutions to **Milord II** development.

In the first part of this Chapter we explain three ES application (*Terap–IA*, *Spong–IA*, and *Ens–AI*). *Terap–IA* is presented more extensively than the others. Finally we explain two examples. The firs one deals with fuzzy control and the other with the propagation of belief in bayesian polytrees.

## 6.2 Terap-IA

Terap-IA is a medical application for pneumonia treatment developed at the *IIIA* by Dr. Pilar Barrufet using **Milord II**. It is a collaboration with the Mataró Hospital directed by Dr. Albert Verdaguer. A Ph.D. Thesis based on *Terap–IA* will be presented soon by Dr. Pilar Barrufet. *Terap–IA* is the natural extension of a previous expert system named *Pneumon–IA* for pneumonia diagnosis (Verdaguer, 1989) developed using *Milord* (Sierra, 1989).

### 6.2.1 Motivation and Goals

The most common cause of mortality related with infectious processes is the pneumonia (it is the sixth death most common cause at EEUU).

The death rate of patients affected by pneumonia that needs hospitalization, is very high. About of 54% of gravely ill patients at the intensive care unit died, and about 20% were old people. In other cases the death rate is about 5.7%. The death rate of patients that do not need hospitalization is lower than those considered above. Despite this, every pneumonia case needs an urgent diagnosis and treatment. Erroneous initial diagnosis can be fatal in some cases. For instance, an initial diagnosis of pneumococic pneumonia in a patient with legionella pneumonia can be fatal because of late adequate treatment.

The goal of *Terap–IA* ES is to deduce the best antibiotic treatment in the case of a pneumonia caused by only one etiologic agent or considering different etiologic hypothesis (definitive diagnosis are about 50%). In the last case we should combine the set of antibiotics corresponding to each germ. Thus it must take into account a set of criterion used in the antibiotic combination.

### 6.2.2 Architecture

We describe the process of obtaining the treatment of pneumonia for a patient having a previous diagnosis (set of possible germs). The goal is to produce an adequate treatment for the case using the previous diagnosis and the data of the patient . The architecture of *Terap–IA* is graphically represented in Figure 6.1.

1. We start with the set of groups of antibiotics[1] used for the pneumonia treatment. Initially we consider that the uncertainty values of these groups of antibiotics are all *true*.

2. *General conditions:* A set of general conditions obtained from the patient is used to filter those groups of antibiotics. Filter these groups consist in changing its uncertainty value by means of rules (remember that initially

[1] The list of 27 groups of antibiotics is the following: Quinolones, Tetraciclin, Tetraciclin Retard, Cotrimoxazol, Sulfamids, Vancomicine, Teicoplanine, Aminoglucocids, Metronidazol, Clindamicine, Carbapenems, Isoniacida, Rifampicina, Etambutol, Pirazinamida, Anfotericina B, Aciclovir, Ganciclovir, Vidarabina, Ribaravina, Amantadine, Rimantadine, Penicilline, Macrolids, Betalactamases Inhibitory, Cefalosporines, and Monobactams.

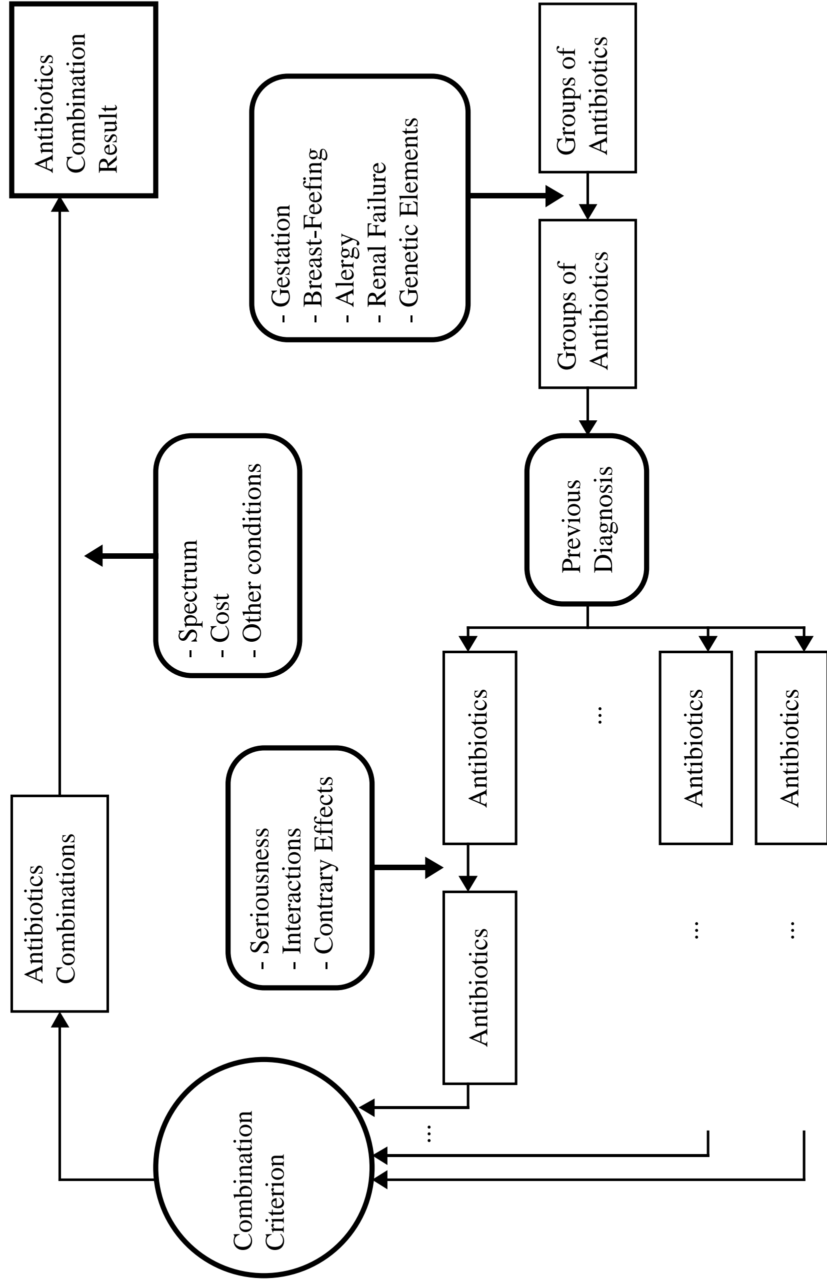


Figure 6.1: Architecture of *Terap–IA* application.

all the groups have the value *true*). The order of filtering is that of the following items[2].

(a) *Gestation*

(b) *Breast–feeding*

(c) *Allergy*

(d) *Renal Failure*

(e) *Genetic Elements*

For instance the expert states that *If the patient has renal failure then the certainty value of Aminoglucocids decreases from sure to possible.* The result of this filtering is the same groups of antibiotics that we got initially but with their truth–values now adequated to the current case.

3. The starting point is a set of germs that are selected from a previous initial diagnosis of the patient. These germs are the possible causes of the disease[3]. The user selects a set of these germs.

4. For each germ selected we filter the above groups of antibiotics:

   (a) *Bacterian Sensibility:* Given a germ and the groups of antibiotics with its certainty value for the germ, the process of filtering selects the groups of antibiotics that can be used in a treatment for this germ.

   (b) *Seriousness:* For each group of antibiotics the seriousness of the patient determines the concrete antibiotics of that group that are adecuate for the treatment.

   (c) *Interactions:* Filter the concrete antibiotics that have interactions with other treatments.

   (d) *Contrary Effects:* Filter the concrete antibiotics taking into account the contrary effects of other treatments.

   The result of this phase is a set of concrete antibiotics with a truth–value associated for each germ considered.

5. When the system has a set of antibiotics one for each germ considered then:

   (a) *Antibiotic Combination:* It combines the treatment for each germ returning a global treatment (a combination of antibiotics).

[2]The filtering of the groups of antibiotics is not dependent of the germ selected.

[3]The list of the 24 germs is the following: Mycoplasma, Coxiella Burnetii, Chlamydia Psitacii, Chlamydia Pneumoniae, Legionella Pneumophila, Pneumococcal Pneumonia, Anaerobis, Enterobacteria, Influenza Virus, branh, Pseudomonas, Meningococcus, S Pyog, S Aurea, Aspergilus, Crip, Nocar, Cytomegalovirus, Varicela–zoster Virus, Herpes Simplex Virus, Eptein–Barr Virus, Respiratory Syncitial Virus, Adenovirus and Haemophilus Influenzae.

(b) *Antibiotic Filtering:* Finally the final treatment is filtered taking into account the spectrum of the antibiotics, its cost and other considerations.

   i. *Spectrum*
   ii. *Cost*
   iii. *Other Conditions*

6. Finally the answer is a combination of antibiotics useful to treat the germs selected and adapted to the particular conditions of the patient.

### 6.2.3 Implementation

In this Section we comment relevant characteristics of the code of *Terap–IA*. For a complete example of the code of *Terap–IA* please go to the Section C.2.

The modular structuration of the treatment problem follows the conceptual one given in Figure 6.1. We give a simple example of filtering of groups of antibiotics and the expansion to concrete antibiotics for a given germ. Finally we present briefly the antibiotic combination.

#### Filtering

In this Section we explain briefly the kind of filtering used in this application. Figure 6.2 shows the modular structure of the filtering explained above. Notice how the modular hierarchy is declared. The module *ABS_1* contains all the groups of antibiotics with certainty value *true*. The module *Gestation (Gestacio*[4]*)* is the first filter. It exports the groups of antibiotics filtered by gestation considerations. The father of this module is *Breast–feeding (Lactancia)* and finally *Allergy (Alergia)*.

If we would follow the modular structure we would find the modules *Renal Failure* and *Genetic Elements*. We explain now the module *Renal Failure* as an example of filtering.

Consider the declaration of this module given in Figure 6.3. This module is declared as a refinement of the module *ABS*. The purpose of this refinement operation is only for the inheritance of the dictionary of that module. There is no information hiding.

This module import nothing from the user (only uses the information of its submodules). It exports the groups of antibiotics. The submodules of *Renal Failure* are the module *Allergy* (another filter) and the module *Anam*. Notice that this module is refined with another encapsulated module declaration. The purpose of this refinement is to hide all the facts exported by the module *Anam* but the fact *renal_failure*.

The evaluation type of the module is eager. It asks then for all the facts exported by its submodules, the filtered groups of antibiotics from *Allergy* and the fact *renal_failure*.

[4]We give the english translation of these module names because this application has been written in Catalan.

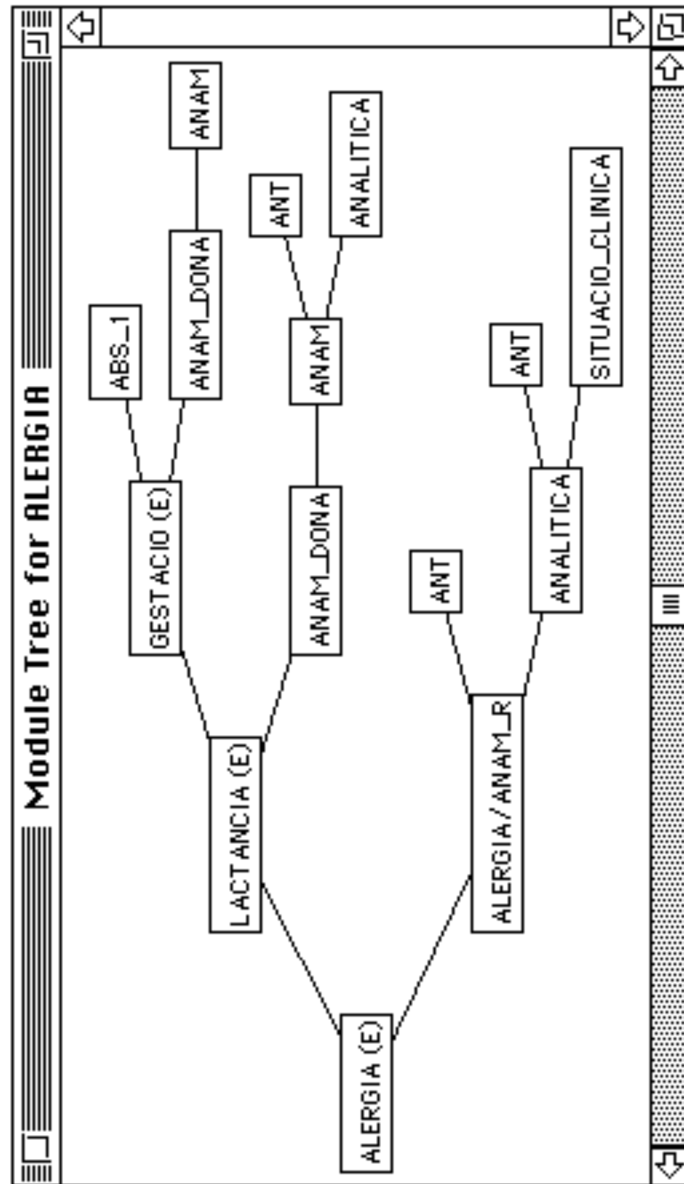


Figure 6.2: Example of filtering.

The rules of this module give a certainty value to a set of groups of antibiotics. Notice that the role of the only metarule of the module is to give a value to the groups of antibiotics that has not been deduced by the module.

M001 **If K**(x/$y,$c) **and NP**($y) **then conclude K**($y,$c)

The meaning of this metarule is the following: given any fact of the submodule $x$ ($x/y$) with any value $c$, such that $y$ has not been deduced by the current module ($NP(\$y)$)[5], then we give to the fact $y$ of the module the value $c$.

This facts maintains the same value that the given by the submodule $x$ (that corresponds to the module *Allergy*). All the filtering modules follows this kind of structure.

```
Module renal_failure:ABS =
    Begin
    Module x = allergy
    Module anam_R= anam:
        Begin
        Export renal_failure
        End
    Export quinol, tetras_1, tetras_2, cotri, sulfas, vanco, teico,
        amino, metro, clinda, carbapen, INH, RFM, ETM, PZ,
        anf_B, ACV, GCV, ARA_A, RBV, AMD, RMD, peni,
        macrol, b_lactam_inh, cef, monobac
    Deductive knowledge
        Rules:
        R001 If anam_R/renal_failure
            then conclude no(tetras_1) is s
        R002 If anam_R/renal_failurel then conclude tetras_2 is p
        R003 If anam_R/renal_failure then conclude amino is p
        R004 If anam_R/renal_failure then conclude anf_B is p
        R005 If anam_R/renal_failure then conclude AMD is p
        End deductive
    Control knowledge
        Evaluation Type: eager
        Deductive Control:
        M001 If K(x/$y,$c) and NP($y) then conclude K($y,$c)
        end control
    End
```

Figure 6.3: Module Renal Failure.

After the filtering of the groups of antibiotics for each germ, we find the concrete antibiotics for the treatment. In Figure 6.4 there is a simple case of treatment for the germ *Pneunomia Mycoplasma*.

[5] The metapredicate $NP$ means $\neg P$ (see Section 5.4.2).

This module inherits the facts of the dictionary from the module *Antimicrobians*. It exports concrete antibiotics. The submodules of this module are the antecedent of the patient *Antecedents* and the last filter *Renal_failure*. Notice that this module is refined in order to only select the groups of antibiotics that are useful for this germ (bacterian sensibility). The rules determine the values for the concrete antibiotics exported.

### Antibiotic Combinations

The initial diagnosis is a set of possible germs. The system deduces a treatment for each germ of the initial diagnosis. We must combine these different treatments in order to give a final treatment (a more complete code is given in Section C.1).

The antibiotic combination consists in two generic modules. The generic module *Build_combinations* produces a set of valid combinations following a set of medical criterion. The generic module Remove_combinations finally simplifies the result of the module *Build_combinations*. These modules are based on metarules. Let us comment one metarule of each generic module.

The first one is a metarule of the generic module *Build_combinations*. This module is instantiated by two modules that represent treatments. Consider two treatments exported by the submodules $x$ and $y$.

M004 **If K**(X/$x,int($tc11,$tc12)) **and K**(Y/$y,int($tc21,$tc22))
**and atom**($x) **and atom**($y) **and diff**($x,$y)
**and no**(subsumeix($x,$y)) **and no**(subsumeix($y,$x))
**and no**(espectre_equivalent($x,$y))
**then conclude WK**(($x plus $y)
,**and**(int($tc11,$tc12),int($tc21,$tc22)))

The above metarule has the following meaning: Given two treatments $X/x$ and $Y/y$ where $x$ and $y$ are simple treatments, that is, antibiotics (atom), and they are different (diff). Furthermore if they have no subsumption relations (in the medical sense of subsumption), then we can conclude a combination of the two antibiotics ($x$ plus $y$). Notice that the conclusion is *weak* ($WK$). That means that this conclusion is not definitive yet and it can be changed by other metarules.

The next example is a metarule of the generic module Remove_combinations.

M006 **If K**(($x plus $y),$V) **and** belongs_to($y,administracio_oral)
**and** belongs_to($x,administracio_parenteral)
**then conclude K**(($x plus $y) ,int(gp,s))

The above metarule considers a combination of two antibiotics ($x$ plus $y$). If they differ in their administration (oral or parenteral) then this combination is removed by attaching to it the value *unknown*.

*Terap–IA* has about 150 modules, 2000 facts, 600 rules, and 200 metarules. It can be considered a big application and it is in a state of validation.

```
Module pneumonia_mycoplasma_treatment: antimicrobians =
    Begin
    Inherit antecedents
    Inherit clinic_situation
    Open renal_failure:
        Begin
        Export quinol, tetras_1, tetras_2, macrol
        End
    Export cipro, oflox, tetras_ac_rap, doxi, doxi_DI, cotri_DB,
        cotri_DI, vanco_tract, teico_tract, amika, genta, metro_tract,
        clinda_DB, clinda_DA, imip, RFM_DA, GCV_tract, ACV_DB,
        ACV_DA, ARA_A_tract, RBV_tract, AMD_DB, AMD_DA,
        RMD_tract, peni_procaina, peni_G_Na, peni_G_Na_DA,
        peni_amp_espectre, cloxa, ampi, amoxi, eritro_DB, eritro_DA,
        roxi, amoxi_clav_DB, amoxi_clav_DA, ticar_clav, cefuro_OR,
        cefuro_EV, ceftriax, cefazol, cefra, cefmet, cefoxi, ceftaz, aztreo
    Deductive knowledge
        Rules:
        R001 If quinol then conclude cipro is modp
            "Kobayashi H. Clinical efficacy of ciprofloxacina in the
            treatment of patients with respiratory tract infections in
            Japan. Am J Med 1987 ; 82(40): 169-73"
        R002 If quinol and clinic_situation/tract_OR
            then conclude oflox is llp
        ;; tetraciclines per mycoplasma
        R003 If tetras_1 then conclude tetras_ac_rap is p
        R004 If tetras_2 then conclude doxi is p
        R005 If tetras_1 then conclude doxi_DI is p
        ;; macrolids per mycoplasma
        R006 If macrol then conclude eritro_DB is s
        R007 If macrol then conclude eritro_DA is s
        R008 If macrol and situacio_clinica/tract_OR
            then conclude roxi is mp
        End deductive
    End
```

Figure 6.4: Module Pneumonia Mycoplasma Treatment.

## 6.3 *Spong–IA*

*Spong–IA* is an ES for sponge classification developed at the IIIA by Marta Domingo (Domingo, 1993a; Domingo, 1993b) using **Milord II**. It is a collaboration with the Ecology group at the CEAB and have been directed by Dr. Carlos Sierra and Dr. Iosune Uriz. A Ph.D. Thesis will be presented soon by Marta Domingo.

Biological classification is a task performed usually by sistematists, but speciment identification can also be the work of either an expert or a general scientist, who being a specialist in other fields, needs the identification of a sample as a starting point for his research.

While an expert would search for a sequence of characters inspired by his experience, a novice or a laboratory technician can be prepared to perform the collection of data, but he can still feel lost in the selection of relevant characters for a given case of classification. The knowledge he would need are the rules of thumb of experts in the field, which in fact are adquired only after years of experience. This is the kind of knowledge that an ES is specially suited to model.

In Figure 6.5 there is an example of the modular structure for *Geodia* sponges. In Figure 6.6 there is a partial example of a concrete case of classification.

*Spong–IA* is an application that has approximately 60 modules, 350 facts, 300 rules and 150 metarules. In is starting now the validation step.

## 6.4 *Ens–AI*

*Ens–AI* (Barroso, 1992) is an intelligent tutorial system directed to the diagnosis and orientation assistency in pedagogical processes. This work is directed to the student education and its goal is to obtain the diagnosis and orientation assistance. This ES has been developed by Dr. Clara Barroso of the La Laguna University (Canarian Islands).

The pedagogical knowledge is composed by a set of different domains of knowledge, mainly the psychology, the pedagogy and the teaching knowledge. Educational problems are treated by *multiprofesional teams* composed by educational professionals like psychologist, pedagogues and teachers.

The knowledge on education, or pedagogic knowledge, is composed by a set of interrelated knowledge. *Expert teams* of schools deal with the diagnosis and recommendations for educational problems. They are composed by psychologists, pedagogues and teachers specialized on elementary education. They advice and orient the general teacher to act in front of a given problem, following the psychopedagogical analysis and diagnosis of the problematic student. The kind of information managed by the *expert team* to elaborate their diagnosis is based in three aspects:

**The student:** The social and individual characteristics of the student.

**His training environment:** Going from the internal characteristics of the school to the external features of the family or his social environment.

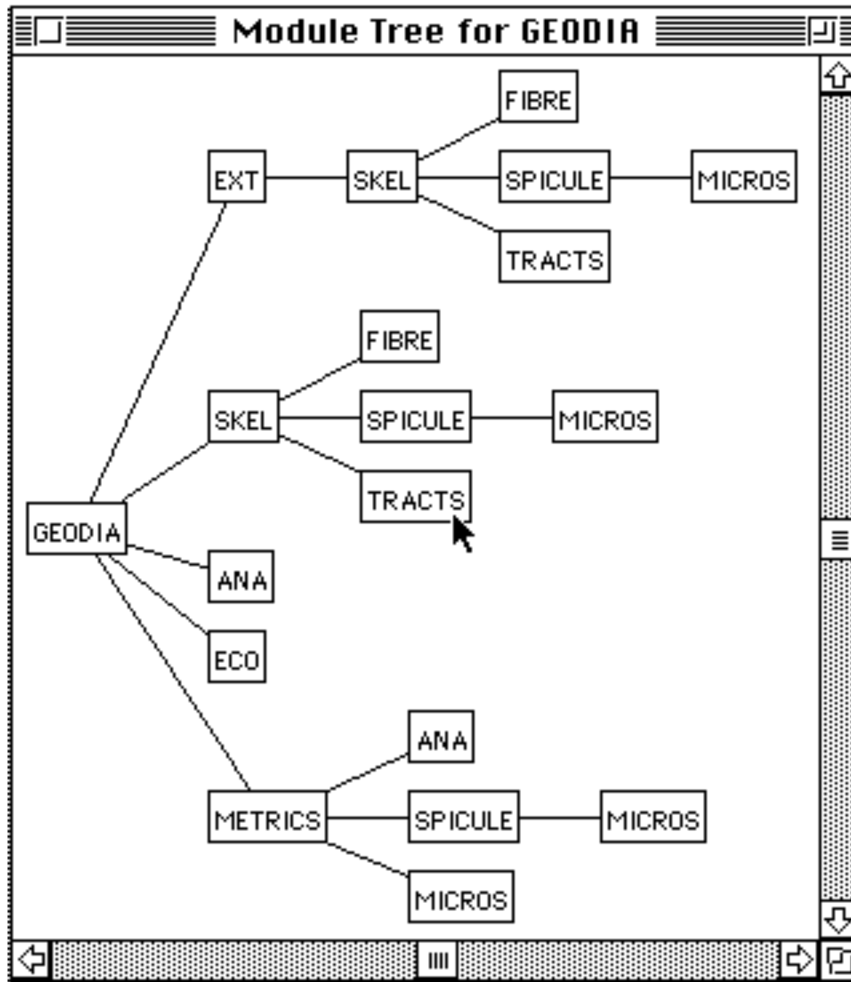


Figure 6.5: Example of module tree.

```
==========================================
Case Name:  Geodia
Date:  2-2-1994 13:49
------------------------------------------
Long Name:  GEOGRAPHICAL LOCATION
Value:  (MEDITERRANEAN)
Module:  CLASSES Name:  GEO
------------------------------------------
Long Name:  PRESENCE OF SPICULES
Value:  S
Module:  SKEL Name:  PRES
------------------------------------------
Long Name:  FIBRES PRESENT IN THE SKELETON
Value:  GP
Module:  SKEL Name:  FIBRE
------------------------------------------
Long Name:  SPICULAR FIBRES OR TRACTS
Value:  GP
Module:  SKEL Name:  TRACTS
------------------------------------------
Long Name:  CHEMICAL COMPOSITION
Value:  (SILICA)
Module:  SKEL Name:  QUIM
------------------------------------------
Long Name:  SPICULE CATEGORIES
Value:  (BOTH_CATEGORIES)
Module:  SPICULE Name:  SIZE
------------------------------------------
Long Name:  NUMBER OF THE SPICULE AXES
Value:  (FOUR ONE)
Module:  SPICULE Name:  AXIS
------------------------------------------
Long Name:  NUMBER OF THE SPICULE ACTINES
Value:  (MORE_THAN_TWO TWO)
Module:  SPICULE Name:  ACTINE
------------------------------------------
Long Name:  TYPE OF MEGASCLERES
Value:  (OXEA TRIAENA)
Module:  SPICULE Name:  MEGAS
==========================================
```

Figure 6.6: Case example.

**His teacher:** His education level, skills, expertise, speciality, attitude and expectations.

The first symptoms of a problem are usually the low profit or the misbehavior of the student. The analysis of the above data allow the experts to make a diagnosis and to propose concrete actions to solve the situation (such as special or alternative education).

The real situation at the school is that *expert teams* have to work with a great number of cases. They concentrate in the most important ones, for instance psychopathologies or special education needs. The *less important* cases (low profit ,misbehavior, etc) have to be treated by the general teacher without specialized advice.

Teachers usually have not the necessary training to cope with this special cases. They have difficulty to analyze with objectivity the result of their actions. All these facts have motivated the development of an expert system thougth as a tool to advice teachers in their educational task.

*Ens–AI* has about 60 modules, 400 facts, 500 rules and 60 metarules.

## 6.5 Fuzzy Control Example

The first exercise proposed to show the expresivity power of **Milord II** is a simple problem of fuzzy control. Fuzzy control methods rely on the knowledge of a set of rules that link, at a symbolic level, the controller inputs to outputs.

The problem to be solved is the regulation of the level in the second of two coupled tanks as shown in Figure 6.7. Laurent Foulloy introduced this example in (Foulloy, 1993) and we will use the fuzzy inference rules proposed in this paper.

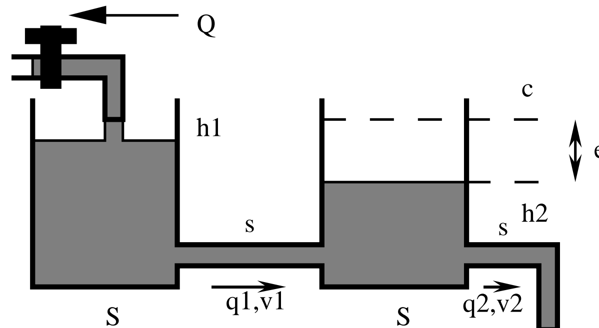


Figure 6.7: Coupled tanks example.

We program the modules of **Milord II** that represent the controller processes of the above control problem. In the Figure 6.8 we can see all the parts of this regulation example. The complete code of this example is given in the Section C.2.

**Simulation Process:** We need to simulate the variation relative to the time of the level of both tanks $h_1$ and $h_2$ given a flow $Q$. The controller acts by intervals of time. Given the sampling period $t_s$ the controller needs to know the level of the second tank at time $kt_s$ , $h_2(kt_s)$, and the first derivative of that level $\frac{dh_2(kt_s)}{dt}$. With this data the controller computes the new flow $Q_{k+1}$ until time $(k+1)t_s$.

**Controller:** The controller is a process with three components:

1. *Fuzzification.* The input of the controller is the actual level in the second tank and its first derivative. First the controller needs to translate these values to qualitative values.
2. *Fuzzy Inference.* These qualitative values can fire a set of rules that represents the fuzzy control. These rules return a new qualitative value for the flow.
3. *Defuzzification.* Finally we must translate the resultant qualitative value of the new flow to the first tank to a physical value.

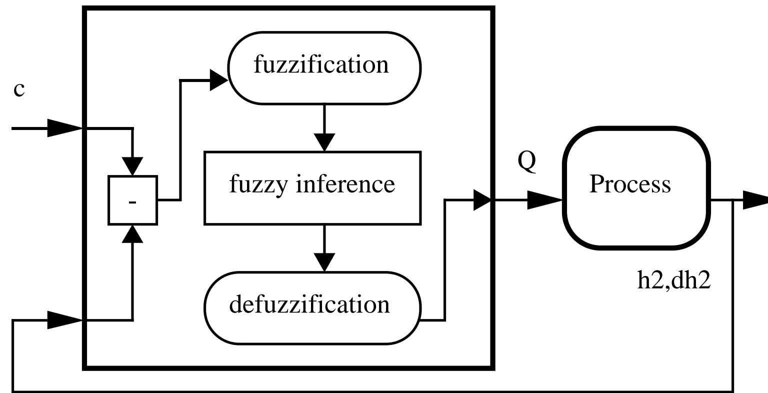


Figure 6.8: Scheme of the process.

### 6.5.1 Simulation Process

Given two identical tanks with section $S$ and the coupling tubes with section $s$ it is easy to see that the levels in the tanks are:

$$\begin{cases} S\frac{dh_1}{dt} = Q - sv_1 \\ S\frac{dh_2}{dt} = sv_1 - sv_2 \end{cases}$$

where

$$sv_1 = \frac{\rho g}{R}(h_1 - h_2)$$

and

$$sv_2 = \frac{\rho g}{R} h_2$$

Normalizing for $\frac{\rho g}{R}$:

$$\begin{cases} \bar{S}\frac{dh_1(t)}{dt} = \bar{Q} - h_1(t) + h_2(t) \\ \bar{S}\frac{dh_2(t)}{dt} = h_1(t) - 2h_2(t) \end{cases}$$

Finally,

$$\begin{cases} \bar{S}^2\frac{d^2h_2(t)}{dt^2} + 3\bar{S}\frac{d^2h_2(t)}{dt^2} + h_2(t) = \bar{Q} \\ h_1(t) = \bar{S}\frac{dh_2(t)}{dt} + 2h_2(t) \end{cases}$$

and

$$\begin{cases} h_2(t) = c_1e^{p_1t} + c_2e^{p_2t} + \bar{Q} \\ \frac{dh_2(t)}{dt} = c_1p_1e^{p_1t} + c_2p_2e^{p_2t} \\ h_1(t) = c_1(2+\bar{S}p_1)e^{p_1t} + (2+\bar{S}p_2)c_2e^{p_2t} + 2\bar{Q} \end{cases}$$

where $p_1 = -\frac{3+\sqrt{5}}{2S}$ and $p_2 = \frac{-3+\sqrt{5}}{2S}$. Considering the initial conditions $h_1(t_a) = h_{1,a}$ and $h_2(t_a) = h_{2,a}$

$$c_1 = \frac{h_{2,a} - c_2e^{p_2t_a} - \bar{Q}}{e^{p_1t_a}}$$

$$c_2 = \frac{h_{2,a}(2+\bar{S}p_1) - h_{1,a} - \bar{Q}\bar{S}p_1}{\bar{S}(p_1 - p_2)e^{p_2t_a}}$$

Given the initial levels of the tanks $h_{1,(k-1)t_s}$ and $h_{2,(k-1)t_s}$ and the flow $\bar{Q}$ in the interval of time $[(k-1)t_s, kt_s]$ we can calculate the new levels in the first tank and its derivative at time $kt_s$:

$$\begin{cases} h_2(kt_s) = c_1e^{p_1kt_s} + c_2e^{p_2kt_s} + \bar{Q} \\ \frac{dh_2(kt_s)}{dt} = c_1p_1e^{p_1kt_s} + c_2p_2e^{p_2kt_s} \end{cases}$$

We use a set of Lisp functions to simulate this process.

### 6.5.2 Controller

Now we introduce in detail the three components of the controller: the fuzzification, the fuzzy inference and the defuzzification. The implementation of this fuzzy controller is very simple. It consists in using the function attribute of the facts for implementing the fuzzificator and the defuzzificator, and the **Milord II** rules for the fuzzy inference.

We use three modules, that is, the fuzzificator, the defuzzificator and the controller (see the Figure 6.9).

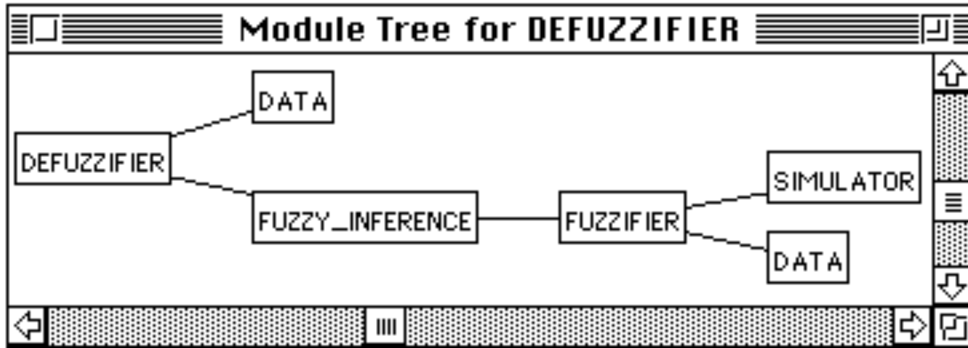


Figure 6.9: Fuzzy control modules.

### Fuzzification

Fuzzification is a numeric to qualitative interface. The numerical value of the error (in this example the difference between the level at the second tank $h_2$ and the reference level $c$) and its derivative are translated to a set of qualitative values. In our case we use a set of symbols *P0*, *PS*, *PM* and *PL* standing respectively for the qualitative constants *Positive Zero*, *Positive Small*, *Positive Medium* and *Positive Large*. The symbols *N0*, *NS*, *NM* and *NL* represent the same for negative values.

In the Figure 6.10 we can see that these symbols are represented as fuzzy sets. The characteristic function is a trapezoid of dimensions *width* and *slope*.

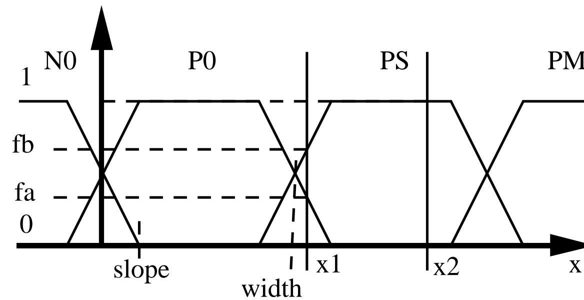


Figure 6.10: Fuzzification process.

In the example of the Figure 6.10 we can see that given a numeric value $x_1$ the fuzzification function returns the value $f_a$ for the symbol *P0* and the value $f_b$ for the symbol *PS*. $f_a$ and $f_b$ are real numbers between 0 and 1[6].

$$f_a = \frac{1}{2} - \frac{1}{2slope}(x_1 - width) \text{ and } f_b = \frac{1}{2} + \frac{1}{2slope}(x_1 - width)$$

[6]Notice that with this representation the fuzzification of a number only returns one or two symbolic values.

In our example these real values are translated to linguistic terms. The linguistic terms used in this example are: impossible, few_possible, sligh_possible, possible, quite_possible, very_possible and sure[7].

In the following module example we can see that the module *fuzzifier* imports the value of the second tank level and its derivative, and exports its qualitative values (*e* and *Var_e*).

We represent these fuzzy sets by means of enumerated facts of type *Q_domain*. The function attribute of these facts are used to make these transformations.

```
Module Fuzzifier =
    Begin
        Inherit Simulator
        Inherit Data
        Export e, Var_e
        Deductive knowledge
            Dictionary:
            Types:
                Q_domain = (PL, PM, PS, P0, N0, NS, NM, NL)
            Predicates:
            e = Name: "Qualitative Value of e"
                Type: Q_domain
                Function:
                (lambda ()
                    (let* ((slope (fact_value Data/s))
                        (terms (type e))
                        (ratio (* (- (division (fact_value Simulator/h2b)
                            (fact_value data/reference))
                                1) 5))
                        (ling_terms (linguistic_terms))
                        (width (fact_value Data/w))
                            (Num_terms (length terms)))
                            ...
            Var_e = Name: "Qualitative Value of Var_e"
                Type: Q_domain
                Function:
                            ...
    End deductive
End
```

For instance, given *slope* = 2.5 and *width* = 5 (triangular case) the fuzzification of 6 returns that *P0 is sligh_possible* and *PS is quite_possible*.

### Fuzzy Inference

Given the qualitative values of the error and its derivative we can use the Table 6.1 (Vicar-Whelan, 1976) to find the resultant value of control $Q$.

[7] We consider that the linguistic terms are equidistant into the interval $[0, 1]$.

| $\epsilon/d\epsilon$ | NL | NM | NS | N0 | P0 | PS | PM | PL |
|---|---|---|---|---|---|---|---|---|
| PL | P0 | PS | NS | NL | NL | NL | NL | NL |
| PM | PS | N0 | NS | NM | NM | NM | NL | NL |
| PS | PM | PS | N0 | NS | NS | NS | NM | NL |
| P0 | PM | PM | PS | P0 | N0 | NS | NM | NM |
| N0 | PM | PM | PS | P0 | N0 | NS | NM | NM |
| NS | PM | PL | PS | PS | PS | P0 | NS | NS |
| NM | PL | PL | PM | PM | PM | PS | P0 | P0 |
| NL | PL | PL | PL | PL | PL | PL | P0 | P0 |

Table 6.1: Mac Vicar–Wheland's initial set of rules.

The qualitative values of the error and the qualitative values of its derivative will produce a set of qualitative values. In our case we represent this table as a set of **Milord II** rules, where the facts used are enumerated facts.

**Module** Fuzzy_Inference =
**Begin**
**Module** F = Fuzzifier
**Export** Var_u
**Deductive knowledge**
**Dictionary:**
**Types:**
Q_domain = (PL, PM, PS, P0, N0, NS, NM, NL)
**Predicates:**
Var_u = **Name:** "Qualitative Action"
**Type:** Q_domain
**Rules:**
*R001* **If** F/e **int** (PL) **and** F/Var_e **int** (NL)
**then conclude** Var_u = (P0) **is** sure
...
*R035* **If** F/e **int** (P0) **and** F/Var_e **int** (NS)
**then conclude** Var_u = (PS) **is** sure
*R036* **If** F/e **int** (P0) **and** F/Var_e **int** (N0)
**then conclude** Var_u = (P0) **is** sure
...
*R043* **If** F/e **int** (PS) **and** F/Var_e **int** (NS)
**then conclude** Var_u = (N0) **is** sure
*R044* **If** F/e **int** (PS) **and** F/Var_e **int** (N0)
**then conclude** Var_u = (NS) **is** sure
...
*R064* **If** F/e **int** (NL) **and** F/Var_e **int** (PL)
**then conclude** Var_u = (P0) **is** sure
**End deductive**
**End**

These 64 rules produce the qualitative output of the control. Notice that in

our system we can introduce weighted rules (in this example all the rules are *sure*) allowing to use richer control strategies.

As shown above the result of a fuzzification process is one or two terms. Taking into account that the rules have as premises the error and its derivative, the output of the fuzzy inference can be one of four terms as maximum (in this example).

For instance, given the value $e = P0$ very_possible *and PS* sure and $Var_e = N0$ very_possible *and NS* possible, the four rules *R035*, *R036*, *R043* and *R044* are fired (see the code example). The result for *Var_u* is NS *very_possible*, N0 *possible*, P0 *possible* and PS *slightly_possible*.

**Defuzzification**

The terms obtained in the fuzzy inference have to be deffuzified in order to obtain a real number representing the physical parameter of control. To do that we use a simple deffuzification method consisting in the computation of the gravity center[8] of the resultant function (see the Figure 6.11).

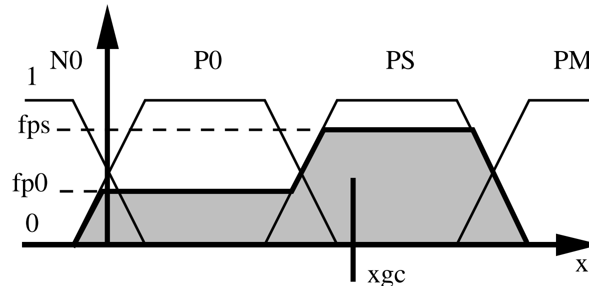


Figure 6.11: Defuzzication by mean of the gravity center.

The defuzzification of the above example *P0 is sligh_possible* and *PS is quite_possible* (for number 6) returns 5.428. Obviously the error has an inverse relation with the number of linguistic terms. The implementation of this module (*defuzzifier*) is similar to the *fuzzifier* module.

### 6.5.3 Results

We run an example to show the whole process. We use $slope = 2.5$, $width = 5$, $\bar{S} = 50$ and the *reference level* of 800. Initially we consider that the two tanks are empty, and there is no flow into the first tank.

In the Figure 6.12 we represent the level in the second tank, the flow into the first tank and the control actions. We can observe an small error in the result of the level in the second tank (805 instead of 800). This is produced by the small number of linguistic terms used.

[8] There are several methods for the deffuzification process, for instance those given in (Berenji, 1992).

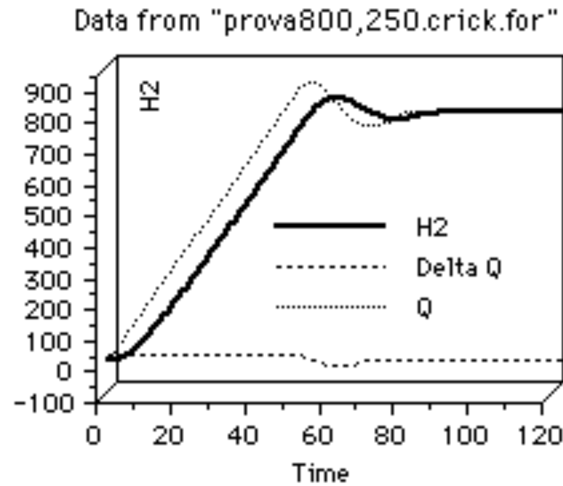


Figure 6.12: Results for $h_2$, $dQ$ and $Q$.

In the Figure 6.13 we can see a detailed graph of the evolution of the control actions. The stabilization is finally performed by mean of small alternatively positive and negative actions of control.

Finally in the Figure 6.14 we can see the phase plane result of the simulation. Observe how the absolute value of the error and its variation decreases to the origin.

## 6.6 Propagation Rules for Polytrees

Here we present an application whose goal is to obtain a declarative algorithm for the propagation of belief in bayesian polytrees. This application was developed by Jose Carlos Ortiz and proposes minimal modifications to **Milord II** in order to reach this goal. Finally bayesian reasoning has been added to the inference machinery of **Milord II**.

### 6.6.1 Introduction

We start from the Belief Updating by Network Propagation of Judea Pearl (Pearl, 1988). A polytree is a singly connected network, namely, no more than one path exists between any two nodes.

Let considers a typical node having $m$ children, $Y_1, \ldots, Y_m$, and $n$ parents, $U_1, \ldots, U_n$ as in Figure 6.15.

The belief distribution of variable $X$ can be computed if the three following types of parameters are made available:

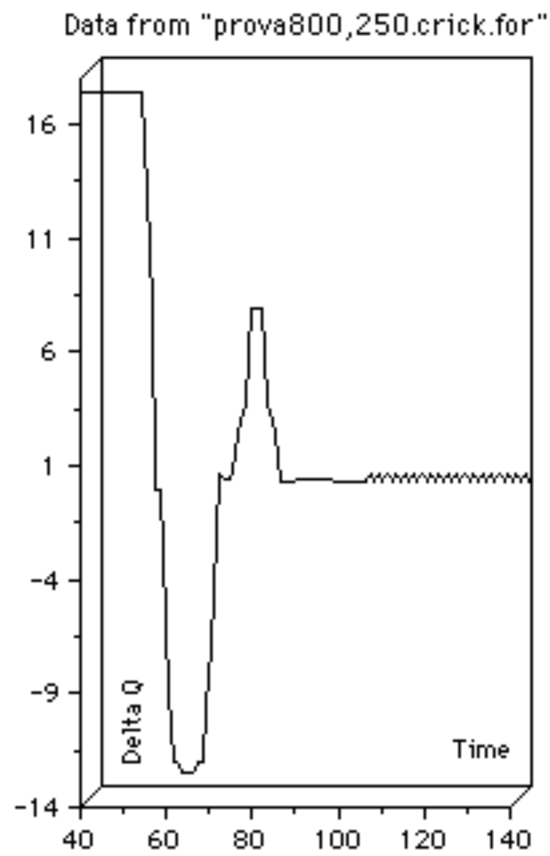


Figure 6.13: Detailed $dQ$.

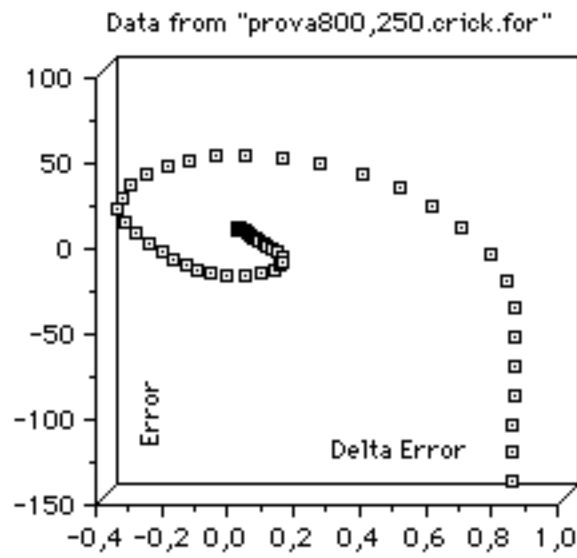


Figure 6.14: Phase plane result.

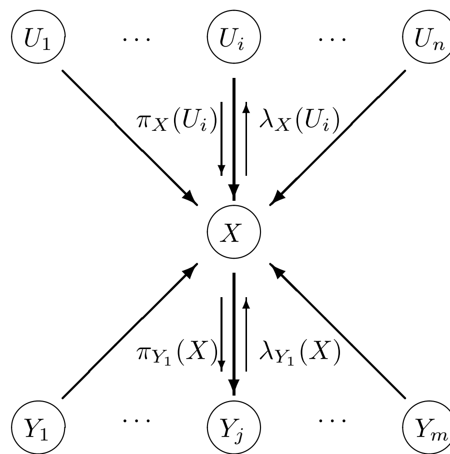


Figure 6.15: Node example.

1. The current strength of the causal support $\pi$ contributed be each incoming link $U_i \rightarrow X$.

2. The current strength of the diagnostic support $\lambda$ contributed by each outgoing link $X \rightarrow Y_j$.

3. The fixed conditional–probability matrix $P(x|u_1, \ldots, u_n)$ that relates the variable $X$ to its immediate parents.

Local belief updating can be accomplished in three steps, to be executed in any order.

**Belief Updating:** Node $X$ inspects the messages $\pi_X(u_i), i = 1, \ldots, n$ communicated by its parents (causal support) ant the messages $\lambda_{Y_j}(x), j = 1, \ldots, m$ communicated by its children (diagnostic support) and updates its belief measure to

$$BEL(x) = \alpha \lambda(x) \pi(x)$$

where

$$\lambda(x) = \prod_{j=1}^{m} \lambda_{Y_j}(x)$$

and

$$\pi(x) = \sum_{u_1, \ldots, u_n} P(x|u_1, \ldots, u_n) \prod_{i=1}^{n} \pi_X(u_i)$$

and $\alpha$ is a normalizing constant.

**Bottom–up propagation:** Using the messages received, node $X$ computes new $\lambda$ messages to be send to its parents. For instance, the new message $\lambda_X(u_i)$ that $X$ sends to its parents $U_i$ is computed by

$$\lambda_X(u_i) = \beta \sum_{x} \lambda(x) \sum_{u_k : k \neq y} P(x|u_1, \ldots, u_n) \prod_{k \neq y} \pi_X(u_k)$$

**Top–down propagation:** Each node computes new $\pi$ messages to be send to its children. For instance, the new $\pi_{Y_i}(x)$ message that $X$ sends to its child $Y_j$ is computed by

$$\pi_{Y_j} = \alpha \left[ \prod_{k \neq j} \lambda_{Y_k}(x) \right] \sum_{u_1, \ldots, u_n} P(x|u_1, \ldots, u_n) \prod_{i=1}^{n} \pi_X(u_i)$$

Boundary conditions:

1. *Root nodes:* if $X$ is a node without parents, we set $\pi(x)$ equal to the prior probability $P(x)$

2. *Anticipatory nodes:* if $X$ is a childness node that has not been initiated, we set $\lambda(x) = (1, 1, \ldots, 1)$

3. *Evidence nodes:* if evidence $X = x'$ is obtained ($X$ being any node in the network) we set $\lambda(x) = (0, \ldots, 0, 1, 0, \ldots, 0)$ with 1 at the x'–th position.

### 6.6.2 Implementation over Milord II

We use the object level and the meta level of **Milord II** in a particular form to deal with this kind of application. At the object level we only represent the causal polytree, nodes and links between nodes. We do not use the inferential mechanisms provided by this level. We use the meta level to the belief propagation along the causal polytree.The complete code for this example is given in Section C.3.

#### Object Level

At the object level we represent the nodes as predicates and the links between nodes as rules. The example of application $n = m = 3$, $U_1 = A$, $U_2 = B$, $U_3 = C$, $X = D$, $Y_1 = E$, $Y_2 = F$ and $Y_3 = G$.

**Nodes:** The nodes of the polytree are facts of type array. We declare all the nodes like the following example for $A$:

A =
  **name:** ”A”
  **type: array**[$a_0$,$a_1$]

**Pointers:** Because of the premises and the rules of the language **Milord II** are of type logic, we use a kind of predicates we name *pointers* as representational facts of the polytree nodes . We declare a pointer for each element of the tree (represented by a relation *points_to* to the facts declare above). For instance the *pointer* to $A$ is:

A_ptr =
  **name:** ”Pointer to A”
  **type:** logic
  **relation:** points_to A

**Rules:** The rules represents the links among the nodes. In this example the links of the node $D$ with its parents $A$, $B$ and $C$; and its children $E$, $F$ and $G$. We need four rules (using the pointers):

R01: **If** A_ptr **and** B_ptr **and** C_ptr **then conclude** D_ptr **is** $P(d|a, b, c)$
R02: **If** D_ptr **then conclude** E_ptr **is** $P(e|d)$
R03: **If** D_ptr **then conclude** F_ptr **is** $P(f|d)$
R04: **If** D_ptr **then conclude** G_ptr **is** $P(g|d)$

**Prior Probabilities:** For each root node (node with no parents, in this example $A$, $B$ and $C$) we declare a fact that represents its prior probability. For instance, for the root node $A$ the declaration of its prior probability is:

A_prior =
  **name:** ”P(A) Prior probability of A”
  **question:** ”Enter P(A), prior probability for A”
  **type: array**[$a_0$,$a_1$]
  **relation:** prior A

The value by default for prior probability assumes equiprobability.

**Evidences:** Each node with evidence (all the leaves, in this example the nodes $E$, $F$ and $G$) has a fact that represents its evidence.

E_evid =
**name:** "Evidence for E"
**question:** "Enter evidence for E"
**type: array**[$e_0$,$e_1$]
**relation:** evid E

The interface of the only module of this application declares the export interface containing all the nodes of the polytree, and the import interface with each fact of *evidence* and each fact of *prior probability*:

**Export** A, B, C, D, E, F, G
**Import** A_prior, B_prior, C_prior, E_evid, F_evid, G_evid

### Meta Level

At this point we explain a simplified version of the metarules that deal with the evidence propagation. Remember the evaluation strategy *reified* where we execute by only a step of reification.

After the module asked to the user all the import interface, the object level of the module reifies the facts belonging to the import interface (the only that have a value at the moment) and the rules (with a particular format[9]).

**K**(A_prior, $a_{prior}$)
**K**(B_prior, $b_{prior}$)
**K**(C_prior, $c_{prior}$)
**K**(D_evid, $d_{evid}$)
**K**(E_evid, $e_{evid}$)
**K**(F_evid, $f_{evid}$)
**K**(cause((A, B, C), D), $P(d|a, b, c)$)
**K**(cause((D), E), $P(e|d)$)
**K**(cause((D), F), $P(f|d)$)
**K**(cause((D), G), $P(g|d)$)

After this reification step we apply the metarules belonging to the control knowledge of this module. We will only explain some of the metarules in order to make clear this kind of application.

**Declare nodes:** We declare which is a node in the metalevel:

m01 **If** points_to($x_ptr, $x) **then conclude** node($x)

this metarule produces node(A), node(B), ..., node(G).

[9]In order to simplify the explanation we omit the metarule that makes the translation to this form. Notice that the premises and the conclusion of the rules are the facts pointed by the original of the rules.

**Initialize prior probability:** We initialize the root nodes with the prior probability with the metarule:

m02 **If** prior($x_prior, $x) **and** **K**($x_prior, $v)
**then conclude** **K**(pi($x), $v)

For instance, the node $A$ is a root node *prior(A_prior, A)* and has a value entered by the user **K** (A_prior, $a_{prior}$), then the result is:

**K**(pi(A), $a_{prior}$)

**Initialize evidence:** We initialize the nodes with evidence as lambda predicates:

m03 **If** evid($x_evid, $x) **and** **K**($x_evid, $v)
**then conclude** **K**(lambda($x), $v)

For instance, the node $D$ has evidence *evid(D_evid, D)* and has a value entered by the user **K** (D_evid, $d_{evid}$), then the result is:

**K**(lambda(D), $d_{evid}$)

**Lambda propagation:** Calculates lambda messages for nodes with several fathers ($D$ in the example):

$$\lambda_X(u_i) = \beta \sum_x \lambda(x) \sum_{u_k:k \neq y} P(x|u_1, \ldots, u_n) \prod_{k \neq y} \pi_X(u_k)$$

We use the following metarule:

```
m05 If K(cause($list_of_fathers, $child), $matrix) and
        K(lambda($child), $lambda_child) and
        position($father_i, $list_of_fathers, $i) and
        set_of_instances
            ($msg,
                conj (position ($father_k, $list_of_fathers,$k),
                    neg( equal($k,$i))
                    K(pi_msg ($father_k, $child), $msg)),
            $pi_msgs_fathers_minus_i)
    then conclude
    K(lambda_msg($child, $father_i),
        matrix_product
            ($lambda_child,
            transpose
                (matrix_prod*
                    (cartesian_prod*($pi_msgs_fathers_minus_i),
                    reduce_dim($matrix, $i)))))
```

In the case of nodes with only one father ($E$, $F$ and $G$ in the current example):

$$\lambda_X(u) = \beta \sum_x \lambda(x) P(x|u)$$

We use the following metarule:

m06 **If K**(cause($list_of_fathers, $child), $matrix) **and**
 **cardinal**($list_of_fathers, 1) **and**
 **position**($father, $list_of_fathers, $i) **and**
 **K**(lambda($child), $lambda_child)
 **then conclude**
 **K**(lambda_msg($child, $father),
 **matrix_product**($lambda_child, **transpose**($matrix)))

We can create a lambda message from the previously calculated lambda of the node $D$ , **K**(lambda(D), $d_{evid}$), and the rule **K**(cause((D), E), $P(e|d)$). From the previous metarule we conclude:

**K**(lambda_msg(E, D),
 **matrix_product**($d_{evid}$, **transpose**($P(e|d)$)))

Finally the lambda update function,

$$\lambda(x) = \prod_{j=1}^{m} \lambda_{Y_j}(x)$$

The following metarule implements this function:

m07 **If** node($father) **and**
 **set_of_instances**
 ($msg,
 **K**(lambda_msg ($child, $father), $msg),
 $lambda_msgs_children)
 **then conclude**
 **K**(lambda($father), **inner_product**($lambda_msgs_children))

**Pi propagation:** Similarly to lambda propagation.

**Belief update:** Finally the belief update $BEL(x) = \alpha\lambda(x)\pi(x)$ is implemented with the following metarule:

m11 **If K**(lambda($x),$lambda_x) **and**
 **K**(pi($x),$pi_x)
 **then conclude**
 **K**($x, **norm**(**inner_product**($lambda_x,$pi_x)))

We have explained this application to show that with minimal modifications **Milord II** deal with bayesian reasoning. We have introduced the type of facts *array*, the evaluation strategy *reified* and some metapredicates (for instance, position) and functions over arrays. You can see an incomplete example of the resulting code in the following Figure.

```
Module POLYTREE =
    Begin
    Export A, B, C, D, E, F, G
    Import A_prior, B_prior, C_prior, E_evid, F_evid, G_evid
    Deductive knowledge
        Dictionary:
            Types:
                dom_A = (a0, a1)
                dom_B = (b0, b1, b2)
                dom_E = (e0, e1)
                ...
            Predicates:
            A =
                name: "A"
                type: array [dom_A]
            A_prior =
                name: "P(A) Prior probability of A"
                question: "Enter P(A), prior probability for A"
                type: array [dom_A]
                relation: prior A
            A_ptr =
                name: "Pointer to A"
                type: logic
                relation: points_to A
            D =
                name: "D"
                type: array [dom_D]
            E =
                name: "E"
                type: array [dom_E]
            E_evid =
                name: "Evidence for E"
                question: "Enter evidence for E"
                type: array [dom_E]
                relation: evid E
            ...
        Rules:
        R01 If A_ptr and B_ptr and C_ptr
            then conclude D_ptr is
                ((((0.3 0.7) (0.4 0.6) (0.5 0.5))
                ((0.75 0.25) (0.82 0.18) (0.35 0.65))
                ((0.45 0.55) (0.8 0.2) (0.1 0.9)))
                (((0.3 0.7) (0.99 0.01) (1 0))
                ((0.37 0.63) (0.85 0.15) (0.21 0.79))
```

```
            ((0.45 0.55) (0.99 0.01) (0.27 0.73))))
    R02 If D_ptr then conclude E_ptr is ((0.75 0.25) (0.55 0.45))
    R03 If D_ptr then conclude F_ptr is ((0.3 0.2 0.5) (0.1 0.5 0.4))
    R04 If D_ptr then conclude G_ptr is ((0.3 0.6 0.1) (0.5 0.2 0.3))
    end deductive
Control knowledge
    Evaluation Type: reified
    Deductive Control:
    m01 ...
    m02 If points_to ($x_ptr, $x) then conclude node($x)
    m03 If evid($x_evid, $x) and K($x_evid, $v)
        then conclude K(lambda($x), $v)
    m04 If prior($x_prior, $x) and K($x_prior, $v)
        then conclude K(pi($x), $v)
    m05 If K(cause($list_of_fathers, $child), $matrix) and
        cardinal($list_of_fathers, 1) and
        position($father, $list_of_fathers, $i) and
        K(lambda($child), $lambda_child)
        then conclude
        K(lambda_msg($child, $father),
            matrix_product($lambda_child, transpose($matrix)))
    ...
    end control
end
```

## 6.7 Future Applications

**Milord II** has being extended with temporal representation and reasoning capabilities (Vila, 1995). The knowledge representation languange has been enriched to express the temporal reference of facts as well as temporal relations between them. Such a language enrichement is efficiently supported by especialized temporal constraint propagation algorithms. **Milord II**'s temporal extension is founded on a formally studied temporal ontology based on instants and periods as pairs of instants (Vila, 1993b; Vila, 1993c), *states*, *fluents* and *accomplishment events* as temporal entities (Vila, 1993d), pointwise metric temporal constraints as temporal relations (Dechter et al., 1991; Vila, 1993a) and a logic –embodying these representational issues– based on the notion of *temporal token* introduced as an argument, namely *temporal token arguments*. Porc-IA is an ESs based on temporal representation to program the evolution of a pig farm.

## 6.8 Conclusions

We have presented the main applications and examples developed using **Milord II**. Our experience with experts is very satisfactory and we are able to start new applications soon. Experts use intensively the modular decomposition of problems and the generic modules. This allows them to make easier to un-

derstand code (real applications have hundreds of facts and rules). An example of the size of these applications is given in Figure 6.16.

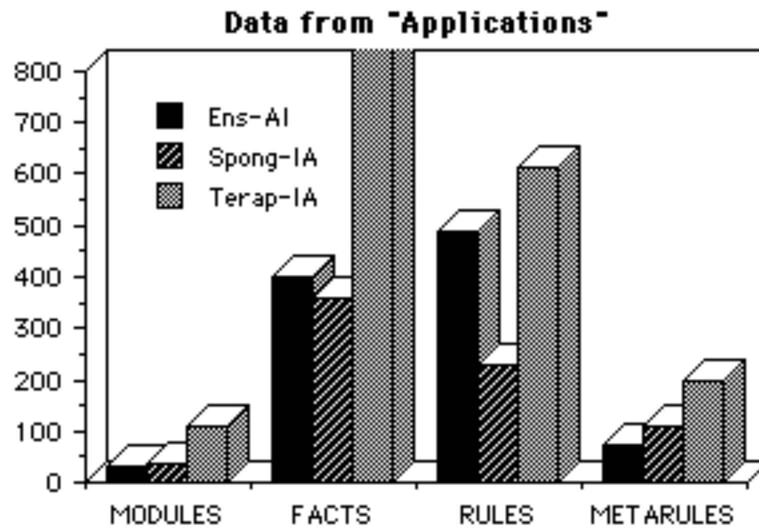


Figure 6.16: Comparations among applications.

We have showed also that **Milord II** is able to represent other kind of problems.

# Chapter 7

# Conclusions

We have presented **Milord II**. It is a language and an architecture to build knowledge–based systems. In this Chapter we present the conclusions of this thesis and the future work on **Milord II**.

The conclusions are summarized in the following items:

**A Modular Language:** All the programming work performed with **Milord II** is based on modules. Modules are the primitive components of the language. The applications programmed with **Milord II** start by structuring the whole problem in a hierarchy of modules. It is a language adapted to programming in the large, that is, to program real applications.

**Milord II** has not global components in the system. Each module contains a complete ES specialized in a part of the whole application. Modules have its own deductive knowledge (dictionary, rules and so on), its own local logic (particular multi–valued logic used to cope with the concrete subproblem) and the local control. Modules have well defined interfaces to interact with the user and the other modules of the system.

**Milord II** modular language is based on modules and generic modules. Generic modules allows us to save code and to make more understandable the code of an application.

A set of operations on modules have been designed to deal with incremental programming of applications. Refinement, contraction and expansion of modules allows the expert to build several versions of modules that are progressively refinements and modifications . All the successive versions of a module are executable entities allowing an incremental validation and testing of the applications.

**Imperfect Knowledge:** We have introduced the local logics of modules of **Milord II**. Expertise implies to deal with imperfect information. The information managed by experts is imprecise and uncertain. A language for ESs must provide the means of expressing easily this kind of information.

Furthermore any good language to express uncertainty is problem dependent.

**Milord II** introduces a familly of multi–valued algebras that are useful to represent uncertainty by means of linguistic terms. The extension of these algebras to intervals of truth–values has been used to deal with imprecision and fuzzy sets.

Local logics have been introduced as a form to adapt the logic to the concrete problem and the techniques to allow the commmunication among modules with different logics have been provided.

**Deduction by Specialization:** The deductive knowledge of the modules of **Milord II** is composed of weighted facts and rules and it has a set of added functionalities that contribute to the practical implementation of ESs.

The inference engine of **Milord II** is based on specialization of KBs. It is a deductive method that provides several improvements on classical inference engines. The search strategy, the communication of the system, the validation and the deductive economy of the system are improved thanks to this mechanism. We have formulated the specialization calculus and presented the practical implementation of an inference engine based on specialization.

**Control:** We have presented the implicit control of **Milord II**. It deals with the economy of deduction and quering to the user (unnecessary rules) and with incomplete knowledge (subsumption). These thecniques takes benefits from specialization.

The inference engine based on specialization allows us to separate the search component and the deductive one. Thanks to this separation we can program different search strategies independently of the deduction.

Each module can contain a local control component based on Horn–like rules. It provides a powerful resource to implement complex deductive behaviours.

**Applications:** Real ESs can be programmed using **Milord II**. The implementation and validation of several ESs are now being finished. The domains of these applications are heterogeneous, from medical applications (*Terap–IA*) to biological classification (*Spong–IA*).

We have showed that **Milord II** is also useful in other kind of applications such as fuzzy control and belief propagation.

## 7.1 Future Work

**Milord II** is the starting point of a language for autonomous agents (Puyol, 1989b; Puyol, 1989a; Puyol, 1990) where fuzzy control and robotics will be

the main topics. We will use the capabilities of specialization to implement knowledge communication among agents.

# Appendix A

# Syntax of Milord II

Syntax of **Milord II**
September 13, 2002
Version 3.4
IIIA-CSIC

## A.1 Notation

The symbols ::=, [, ], | are part of the BNF formalism, as follows:

| | |
|---|---|
| L ::= R | The syntax of L is defined by R |
| [ X ] | An optional item |
| X \| Y | An item from one of the syntactic categories X or Y |

- We write the predefined terminal symbols that are part of the language **Milord II** in **underlined boldface** [1].
- We write user-defined terminal symbols in *italic*. They are always atomic symbols.
- We write non-terminal symbols in normal type face.

[1] Comparison of predefined terminal symbols is case-insensitive.

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

**Note:** Some parts of this syntax are not explained in the text of this book, mainly those refering the temporal extension of **Milord II**. Please refer (Vila, 1995) for a complete explanation.

## A.2 Modular System

| | | |
|---|---|---|
| environ | ::= | moddecl environ \| moddecl |
| moddecl | ::= | **Module** modbind |
| modbind | ::= | *amodid* [**(**[paramlist] **)**][ modop modexpr] |
| | | [**≡** modexpr] |
| modexpr | ::= | bodyexpr [ modop modexpr] |
| modop | ::= | **:** \| **≤** \| **≥** |
| bodyexpr | ::= | *pathid*[**(**[iparamlist] **)**] \| |
| | | **begin** decl **end** \| |
| | | **nil** |
| paramlist | ::= | params [**;** sharing] |
| params | ::= | *amodid* [modop modexpr] \| params **;** params |
| iparamlist | ::= | modexpr \| iparamlist **;** iparamlist |
| decl | ::= | [hierarchy] |
| | | [interface] |
| | | [deductive] |
| | | [control] |
| hierarchy | ::= | **Open** modexpr \| |
| | | **Sharing** patheq \| |
| | | **Inherit** pathid \| |
| | | moddecl \| |
| | | hierarchy hierarchy |
| interface | ::= | [import] |
| | | [export] |
| sharing | ::= | **sharing** patheq |
| pathid | ::= | *amodid* \| *amodid***/**pathid |
| patheq | ::= | pathid **≡** patheq \| pathid **≡** pathid \| |
| | | patheq **;** patheq |
| import | ::= | **Import** predicateidlist |
| export | ::= | **Export** predicateidlist |
| predicateidlist | ::= | *predid* **,** predicateidlist \| *predid* |
| pathpredid | ::= | *predid* \| *amodid***/**pathpredid |

## A.3 Deductive Knowledge

| | | |
|---|---|---|
| deductive | ::= | **Deductive knowledge** |
| | | [**Dictionary:** dictionary] |
| | | [**Rules:** rules] |
| | | [**Inference system:** logcomp] |
| | | **end deductive** |

### A.3.1 Dictionary

| | | |
|---|---|---|
| dictionary | ::= | [typedef] preddef |
| typedef | ::= | **Types:** typebindings |
| typebindings | ::= | *typeid* [**=** [**temporal**] typespec] \| |
| | | typebindings typebindings |
| typespec | ::= | **(**values **)**\| |
| | | predefined \| |
| | | *typeid* \| |
| | | **Array** **[** arraydim **]** \| |
| | | **Set Of** *typeid* \| |
| | | **frame** framespec **end frame** |
| predefined | ::= | **boolean** \| **logic** \| **numeric** \| **class** |
| arraydim | ::= | *typeid* \| |
| | | *numeric* \| |
| | | arraydim **,** arraydim |
| framespec | ::= | *slotid* **:** *typeid* |
| | | framespec framespec |
| preddef | ::= | **Predicates:** predbindings |
| predbindings | ::= | *predid* **=** attributes \| |
| | | predbindings predbindings |
| attributes | ::= | name [question] type [function] [relations] |
| name | ::= | **Name:** *string* |
| question | ::= | **Question:** *string* |
| type | ::= | **Type:** [**temporal**] typespec |
| function | ::= | **Function:** *S-expression* |
| relations | ::= | **Relation:** *relationid pathpredid* \| |
| | | relations relations |
| relationid | ::= | predef-rel-id \| *symbol* |
| predef-rel-id | ::= | **Needs** \| **Needs_true** \| |
| | | **Needs_false** \| **Belongs_to** |

### A.3.2 Rules

| | | |
|---|---|---|
| rules | ::= | rule rules \| rule |
| rule | ::= | ruleid **If** premisse-rule **Then** conclusion-rule |
| | | [*documentation*] |
| premisse-rule | ::= | condition-rule **and** premisse-rule \| condition-rule |

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

condition-rule ::= conditio |
**no** (conditio )

rconclusion ::= form |
elemental ≡ (values )|
**no** (elemental )|
**no** (form *op* form )|
**no** (elemental ≡ (values ))

form ::= elemental | (form *op* form )

elemental ::= *predid* | *varid*

values ::= *symbol* **,** values | *symbol*

values1 ::= *symbol* **between** [ *ltermid* **,** *ltermid*] **,** values1 |
*symbol* **between** [ *ltermid* **,** *ltermid*]

conditio ::= operator (expression, ..., expression )|
pathform | cert-value |
expression operator expression

pathform ::= pathform-s | pathform-c

pathform-s ::= elemental | *amodid*/pathform-s | *varid*/pathform-s

pathform-c ::= (formula )| *amodid*/pathform-c | *varid*/pathform-c

formula ::= (pathform *op* pathform )

expression ::= operator-arit (expression, ..., expression )|
(expression operator-arit expression )|
numeric | elemental | pathpredid | pathvar |
values | values1

S-expression ::= atom | list | S-expression S-expression

list ::= (S-expression )| ()

operator ::= $\leq$ | $\geq$ | $<=$ | $>=$ | ≡ | /= | **int**

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

## A.4 Inference System

| | | |
|---|---|---|
| logcomp | ::= | [lingtermdef] [order]<br>[renaming]<br>[connectives]<br>[infpat] |
| lingtermdef | ::= | **Truth values =** (ltermidlist )\|<br>**Truth values =** (ltermlist ) |
| ltermidlist | ::= | *ltermid* , ltermidlist \| *ltermid* |
| ltermlist | ::= | *ltermid* ≡ (real, real, real, real ), ltermlist \|<br>*ltermid* ≡ (real, real, real, real) |
| cert-value | ::= | *ltermid* \|<br>*number* \|<br>[ *ltermid* , *ltermid*] \|<br>[*number* , *number*] \|<br>(certarrows ) |
| certarrows | ::= | certlist \|<br>(certarrows )certarrows |
| certlist | ::= | *ltermid* \|<br>*number* \|<br>certlist certlist |
| renaming | ::= | **Renaming** lrenames |
| lrenames | ::= | pathid/cert-value ==> cert-value lrenames \|<br>pathid/cert-value ==>cert-value \|<br>S-expression |
| connectives | ::= | **Connectives:**<br>[**Negation** ≡ fundef]<br>[**Conjunction** ≡ fundef]<br>[**Disjunction** ≡ fundef] |
| infpat | ::= | **Inference patterns:**<br>**Modus ponens** ≡ fundef |
| fundef | ::= | S-expression \| fileid \| **luckasiewicz** \|<br>**Zadeh** \| **probabilistic** \| truth-table |
| order | ::= | (ltermid, ltermid) order \|<br>$\lambda$ |
| truth-table | ::= | **Truth table** (arrows) |
| arrows | ::= | (termlist) \| arrows arrows |
| ltermlist | ::= | ltermid \| ltermlist ltermlist |

## A.5 Control Knowledge

| | | |
|---|---|---|
| control | ::= | **Control knowledge** [search] [threshold][deduccnt] [structcnt] **end control** |

### A.5.1 Evaluation Type

| | | |
|---|---|---|
| search | ::= | **Evaluation type:** evaltype |
| evaltype | ::= | **lazy** \| **eager** \| **heuristic** \| **input** \| **reified** |

### A.5.2 Truth Threshold

| | | |
|---|---|---|
| threshold | ::= | **Truth threshold:** cert-value |

### A.5.3 Deductive Control

| | | |
|---|---|---|
| deduccnt | ::= | **Deductive control:** lmrr |
| lmrr | ::= | mrr lmrr \| mrr |
| mrr | ::= | metaid **If** premisse-meta **Then** filters-mrr |
| filters-mrr | ::= | filter-mrr filters-mrr \| filter-mrr |
| premisse-meta | ::= | condition-meta **and** premisse-meta \| condition-meta |
| condition-meta | ::= | mconditio \| **no**(mconditio )\| |
| mconditio | ::= | metapredid (conditionterm , ..., conditionterm ) |
| conditionterm | ::= | operation (conditionterm ,..., conditionterm )\| metafunctid (conditionterm , ..., conditionterm )\| conditio |
| filter-mrr | ::= | **inhibit rules** relation-id pathpredid \| **inhibit rules** pathpredid \| **prune** pathpredid \| **increase** form integer \| **decrease** form integer \| conclusion-meta |
| conclusion-meta | ::= | mconclusion \| **no** (mconclusion ) |
| mconclusion | ::= | metapredid (conclusionterm , ..., conclusionterm ) |
| conclusionterm | ::= | operation (conclusionterm , ..., conclusionterm )\| metafunctid (conclusionterm ,..., conclusionterm )\| form |

### A.5.4 Structural Control

| | | |
|---|---|---|
| structcnt | ::= | **Structural control:** lmre |
| lmre | ::= | mre \| mrx \| lmre lmre |

| | | |
|---|---|---|
| mre | ::= | metaid **If** premisse-meta **Then** filter-mre |
| filter-mre | ::= | **filter** amodidlist \| |
| | | **order** amodidlist **with certainty** cert-value \| |
| | | **Open** **(**conclusionterm **,** ...**,** conclusionterm **)**\| |
| | | **Module** **(**conclusionterm **,** ...**,** conclusionterm **)**\| |
| | | **Inherit** **(**conclusionterm **,** ...**,** conclusionterm **)** |
| amodidlist | ::= | amodid amodidlist \| amodid |
| mrx | ::= | metaid **If** premisse-meta **Then** exception |
| exception | ::= | **definitive solution** predid \| |
| | | **stop** |

# Appendix B

# Proofs

**Proposition B.1** $MP^*_T([a,b],[c,1]) = [T(a,c),1]$

*Proof.* It reduces to find all solutions of the functional inequations

$$I_T(x,z) \geq c$$

being $a \leq x \leq b$. However, given a residuated pair $(T, I_T)$ it is well known that the following relation holds:

$$T(x,y) \leq z \text{ iff } I_T(x,z) \geq y$$

Then, the solution for the first equation is $z \geq T(x,c)$, and taking into account that $x \geq a$, and that $z = 1$ is always a solution, the minimal interval that will contain all the solutions for $z$ is $[T(a,c),1]$.□

## B.1 Proposition

**Proposition B.2** *If $p, q, p_1, \ldots, p_n$ denote literal symbols then the following properties are fulfilled:*

**SR1:** $(p,V) \models (p,W) \Leftrightarrow V \subseteq W$

**SR2:** $(p,V) \models (\neg p,W) \Leftrightarrow N^*_n(V) \subseteq W$

**SR3:** $(p,V),(p,W) \models (p,U) \Leftrightarrow V \cap W \subseteq U$

**SR4:** $(p_i,V_i),(p_1 \wedge \cdots \wedge p_n \rightarrow q, V) \models (p_1 \wedge \cdots \wedge p_{i-1} \wedge p_{i+1} \wedge \cdots \wedge p_n \rightarrow q, W) \Leftrightarrow MP^*_T(V_i,V) \subseteq W$

**SR5:** $MP^*_T(T^*(V_1,\ldots,V_n),W) = MP^*_T(V_1, MP^*_T(V_2,\ldots,MP^*_T(V_n,W)\ldots))$, *if* $W = [w,1]$

Finally we explain the proof of the properties:

**SR1:** Straightforward from the satisfaction relation definition.

**SR2:** Follows from SR1 and the fact that a valuation $\rho$ satisfies $\rho(p) \in V$ if, and only if, $\rho(\neg p) \in N_n^*(V)$.

**SR3:** Straightforward from the satisfaction relation definition.

**SR4:** First we prove the property for the simplest Modus Ponens case, i.e.,

$$\{(p,U),(p \to q,V)\} \models (q,W) \text{ iff } MP_T^*(U,V) \subseteq W$$

By definition of the function $MP_T^*$, $MP_T^*(U,V)$ is the minimal interval containing all the solutions for $\rho(q)$ in the following family of functional equation systems:

$$\begin{cases} \rho(p) = a \\ \rho(p \to q) = I_T(\rho(p),\rho(q)) = b \end{cases}$$

for any $a \in U$, and $b \in V$. Thus, for any model $\rho$ satisfying $\rho(p) \in U$, and $\rho(p \to q) \in V$ and $\rho(q) \in W$, it must be only the case that $\rho(p) \in MP_T^*(U,V)$, and thus $MP_T^*(U,V) \subseteq W$. Moreover, if $U = [x,y]$ and $V = [z,1]$, then $MP_T^*(U,V) = [T(x,z),1]$.

Now property SR4 follows straightforward from the associativity of the t-norm $T$, used to interpret conjunctions, and from the fact that a residuated pair $(T, I_T)$ satisfies the following equality:

$$I_T(T(x,y),z) = I_T(x, I_T(y,z))$$

**SR5:** From proposition 1, it follows that, if $U = [x,y]$ and $V = [z,1]$, then $MP_T^*(U,V) = [T(x,z),1]$. Then, it is easy to see that, due to the associativity of the t-norm $T$, if $V_i = [a_i,b_i]$, for $i = 1,\ldots,n$, then

$$T^*(V_1,\ldots,V_n) = [T(a_1,\ldots,a_n)^1, T(b_1,\ldots,b_n)]$$

and thus, on the one hand

$$\begin{aligned} MP_T^*(V_1, MP_T^*(V_2,\ldots, MP_T^*(V_n,W)\cdots)) &= [T(a_1,\ldots,T(a_n,w)\cdots),1] \\ &= [T(a_1,\ldots,a_n,w),1] \end{aligned}$$

and on the other hand

$$\begin{aligned} MP_T^*(T^*(V_1,\ldots,V_n),W) &= MP([T(a_1,\ldots,a_n),T(b_1,\ldots,b_n)],[w,1]) \\ &= [T(a_1,\ldots,a_n,w),1] \end{aligned}$$

Properties from SR1 to SR4 give us a foundation for the specialization calculus.

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

## B.2 Soundness Theorem

From properties SR1, SR2, SR3 and SR4 of the semantical entailment, it is easy to check that the above specialization calculus is sound.

**Theorem B.1 (Soundness)** *Let $A$ be a sentence and $\Gamma$ a set of sentences. Then $\Gamma \vdash A$ implies $\Gamma \models A$*

*Proof.* The properties SR1-SR4 show that the inference rules are locally sound and complete. So, we need only to show that the axioms are sound to have the proof of the theorem.

1. If $A$ is the axiom AS1, i.e. $A = (\neg\neg p \rightarrow p, [1,1])$ then for every model $M_\rho$, $\rho(p) = N(N(\rho(p))) = N(\rho(\neg p)) = \rho(\neg(\neg(p))) \Rightarrow I(\neg\neg p \rightarrow p) = I(\rho(\neg\neg p), \rho(p)) = 1$. Then, for all $M_\rho$, $M_\rho \models (\neg\neg p \rightarrow p, [1,1])$.
2. If $A$ is the axiom AS2, i.e. $A = (p, [0,1])$, it is the case that every model $M_\rho$ satisfies $\rho(p) \in [0,1]$. Then for all $M_\rho$, we have trivially that $M_\rho \models (p, [0,1])$.
3. If $A$ is the axiom A1, i.e., $A = (true, [1,1])$ then, by definition, for every model $M_\rho$, $\rho(true) = 1 \Rightarrow M_\rho \models (true, [1,1])$.
4. If $A$ is the axiom A2, the proof is analogous to the previous case.

## B.3 Restricted Completeness

### B.3.1 Literal Completeness

It is straightforward to see that our deductive system is not complete. For instance, we have $\{(p \rightarrow q, 1), (q \rightarrow r, 1)\} \models (p \rightarrow r, 1)$ but $\{(p \rightarrow q, 1), (q \rightarrow r, 1)\} \not\vdash (p \rightarrow r, 1)$. It is also the case that the language is not complete for literal deduction in general. For instance, we have $\{(p \rightarrow q, 1), (\neg p \rightarrow q, 1)\} \models (q, 1)$ but $\{(p \rightarrow q, 1), (\neg p \rightarrow q, 1)\} \not\vdash (q, 1)$. However, it can be proved that the system is complete for literal deduction in the context of a restricted language setting, as it will be shown in this section.

#### Previous definitions

**Definition B.1 (Mv-Horn-Rules)** *We define the set Mv-Horn-Rules as the set $\{(p_1 \wedge p_2 \wedge \cdots \wedge p_n \rightarrow q, V) \mid p_i$ and $q$ are atomic symbols, $V = [a, 1]$ is an interval of truth-values of $A_n$ with $a > 0$, and $\forall i, j (p_i \neq p_j, q \neq p_j)\ \}$*

**Definition B.2 (Restricted Language)** *Given the propositional language*

$$\mathcal{L}_n = (A_n, \Sigma, \mathcal{C}, \mathcal{S}_n)$$

*we define a restricted propositional language as:*

$$\mathcal{RL}_n = (A_n, \Sigma, \mathcal{C}, \mathcal{RS}_n)$$

*where* $\mathcal{RS}_n$ = *Mv-Atoms* ∪ *Mv-Horn-Rules*

For any $\Gamma \subset \mathcal{RS}_n$ the next notation will be used:

- $\Gamma = \Gamma_L \cup \Gamma_R$
- $\Gamma_L = \{\gamma \in \Gamma | \gamma$ are mv-Atoms$\}$
- $\Gamma_R = \{\gamma \in \Gamma | \gamma$ are mv-Horn-Rules$\}$
- *Prem*: Is a function that given a rule returns its conditions.
- *Cond*: Is a function that given a rule returns its conclusion.
- $\Gamma_L^A = \{p \in \Sigma | \exists V$ interval of $A_n : (p, V) \in \Gamma_L\}$
- $\Gamma_R^A = \{p \in \Sigma | \exists r \in \Gamma_R : p \in Prem(r)$ or $p = Concl(r)\}$
- $\Gamma^A = \Gamma_L^A \cup \Gamma_R^A$

**Previous Lemmas**

**Proposition B.3** *The inference rules (Weakening, Composition, Negation and Specialization) are locally complete, i.e., they verify the following equivalences:*

1. $(p, V) \vdash (p, W)$ *iff* $(p, V) \models (p, W)$
2. $(p, V) \vdash (\neg p, W)$ *iff* $(p, V) \models (\neg p, W)$
3. $\{(p, V), (p, W)\} \vdash (p, U)$ *iff* $\{(p, V), (p, W)\} \models (p, U)$
4. $\{(p_1, V_1), \ldots, (p_n, V_n), (p_1 \wedge \cdots \wedge p_n \rightarrow q, W)\} \vdash (q, U)$ *iff* $\{(p_1, V_1), \ldots, (p_n, V_n), (p_1 \wedge \cdots \wedge p_n \rightarrow q, W)\} \models (q, U)$

*Proof.* Straightforward from properties SR1-SR5 and from the definition of the four inference rules in Section 3.3.

**Lemma B.1** *Let* $\Gamma_L$ *be a set of Mv-Atoms, and let* $R_1$, $R_2$ *be two sets of mv-Horn-Rules.* $R_1$ *and* $R_2$ *rules have as premises conjunctions of atoms belonging to* $\Gamma_L^A$*, and share the same conclusion, an atom* $p$ *not belonging to* $\Gamma_L^A$*.*

*If* $V_1 = \bigcap\{V'' | \{\Gamma_L, R_1\} \models (p, V'')\}$, $V_2 = \bigcap\{V'' | \{\Gamma_L, R_2\} \models (p, V'')\}$ *and* $W = \bigcap\{V'' | \{\Gamma_L, R_1, R_2\} \models (p, V'')\}$*, then* $W \supseteq V_1 \cap V_2$.

*Proof.* By *reductio ad absurdum*. Suppose that $W \not\supseteq V_1 \cap V_2$. Then $\exists \alpha \in V_1 \cap V_2$ and $\alpha \notin W$. Because of $V_1$ and $V_2$ are minimals, we have that:

- $\alpha \in V_1 \Rightarrow \exists M_\rho$ such that $\rho(p) = \alpha$, $M_\rho \models \Gamma_L$ and $M_\rho \models R_1$
- $\alpha \in V_2 \Rightarrow \exists M_{\rho'}$ such that $\rho'(p) = \alpha$, $M_{\rho'} \models \Gamma_L$ and $M_{\rho'} \models R_2$

We will prove that there always exists a model $M_{\rho''}$ such that $\rho''(p) = \alpha$, $M_{\rho''} \models \Gamma_L$, $M_{\rho''} \models R_1$ and $M_{\rho''} \models R_2$. Define $\rho''(p) = \alpha$, and $\rho''(a) = min(\rho(a), \rho'(a))$, $\forall a \in \Gamma_L^A$. $M_{\rho''}$ easily extends to the Mv-Horn-Rules by the implication function $I_T$. Then, for this model $M_{\rho''}$ we have:

1. $\rho''(p) = \alpha$.
2. $M_{\rho''} \models \Gamma_L$: due to the fact that $\rho'' = min(\rho, \rho')$ over $\Gamma_L^A$.
3. $M_{\rho''} \models R_1$: $\forall r \in R_1$, where $r = (q_1 \wedge \cdots \wedge q_m \rightarrow p, [v_r, 1])$, we have that $M_\rho \models R_1$ implies $\rho(r) \geq v_r$. Given that we work with Mv-Horn rules, i.e. $q_i$ are not negated literals, and the monotonicity property of function $T$, it always holds that:
$$\rho''(q_1 \wedge \cdots \wedge q_n) =$$
$$T(\rho''(q_1), \ldots, \rho''(q_n)) \leq T(\rho(q_1), \ldots, \rho(q_n)) = \rho(q_1 \wedge \cdots \wedge q_n)$$
and, given that $I_T$ is not increasing in the first argument, it always holds that:
$$\rho''(r) = I_T(\rho''(q_1 \wedge \cdots \wedge q_n), \alpha) \geq I_T(\rho(q_1 \wedge \cdots \wedge q_n), \alpha) = \rho(r) \geq v_r$$
that is, $M''_\rho \models R_1$.
4. $M_{\rho''} \models R_2$: Analogously to the previous case.

Summarizing, we have found $M_{\rho''}$ such that $\rho''(p) = \alpha$, $M_{\rho''} \models \Gamma_L$, $M_{\rho''} \models R_1$ and $M_{\rho''} \models R_2$, i.e. $\{\Gamma_L, R_1, R_2\} \not\models (p, W)$ which is in contradiction with the enunciate of the lemma.□

Next Lemma shows that previous deductions over a Mv-Atom $p$ do not restrict the models of Mv-Atoms belonging to premises of other rules concluding the same Mv-Atom $p$. In practical terms, having previous deductions over an atom $r$ means that we know $r$ with an interval of truth values of type $[v, 1]$. Otherwise, if we knew $r$ with a general interval of type $[b, c]$ it could be the case that premises of rules concluding $r$ would be semantically deduced with intervals different of $[0, 1]$. On the contrary, it would not be possible to syntactically deduce them. So, next Lemma allows us, when considering atom deducibility, to only consider those rules that deduce it and not the rules that use it as a premise.

**Lemma B.2**

$$\bigcap\{V'' | \{(p \wedge q_1 \wedge \cdots \wedge q_n \rightarrow r, [a, 1]), (r, [b, 1])\} \models (p, V'')\} = [0, 1]$$

*Proof.* It is sufficient to prove that $\forall \alpha \in [0, 1]$, we can find a model $M_\rho$ such that $\rho(p) = \alpha$ and that $M_\rho \models (p \wedge q_1 \wedge \cdots \wedge q_n \rightarrow r, [a, 1])$ and $M_\rho \models (r, [b, 1])$. Actually, every model $M_\rho$ such that $\rho(p) = \alpha$ and $\rho(r) = 1$, satisfies that $\rho(p \wedge q_1 \wedge \cdots \wedge q_n \rightarrow r) = 1$, and thus $M_\rho \models (p \wedge q_1 \wedge \cdots \wedge q_n \rightarrow r, [a, 1])$ and $M_\rho \models (r, [b, 1])$. □

### B.3.2 Restricted Literal Completeness Theorem

**Theorem B.2 (Restricted Literal Completeness)** *If* $\Gamma \models (p, V)$*, then* $\Gamma \vdash (p, V)$*, provided that* $p \in \Gamma^A$*, where* $\Gamma$ *is such that the following conditions hold:*

1. $\Gamma \subset \mathcal{RS}_n$
2. $\forall r \in \Gamma_R : concl(r) \notin \Gamma^A_L$
3. *The deductive and/or graph associated to* $\Gamma$ *is acyclic.*

*Proof.* Given that the and/or deductive graph associated to $\Gamma$ is acyclic, we can decompose the set $\Gamma^A$ of atomic symbols appearing in $\Gamma$ in a set of disjoint layers. The definition of the layers is the following:

- $S_0 = \{q \in \Gamma^A | \nexists r \in \Gamma_R : Concl(r) = q\}$
- $S_1 = \{q \in \Gamma^A | \forall r \in \Gamma_R : Concl(r) = q \Rightarrow \forall x \in Prem(r), x \in S_0\}$
- ...
- $S_i = \{q \in \Gamma^A | \forall r \in \Gamma_R : Concl(r) = q \Rightarrow \forall x \in Prem(r), x \in S_j,$ being $j < i$ and $\exists r : \exists x \in Prem(r), x \in S_{i-1}\}$
- ...

The proof of the theorem is by induction over the layer number $n$ to which $p$ belongs. Suppose that $V \neq [0, 1]$, otherwise the proof of the theorem is trivial.

The set $\Gamma^A$ is decomposed in layers $\Gamma^A = \bigcup_{i=1,n} S_i$. Because of Lemma 2, in order to deduce $p$ we only need to consider that part of $\Gamma$ containing rules using atoms belonging to layers lower than the layer of $p$. That is, we consider only those rules of $\Gamma$ belonging to the deductive subgraph of $p$.

**Case** $n = 0$**:** In this case $\Gamma$ contains a set of mv-atoms as

$$\{(p, V_i) | i \in I\} \subseteq \Gamma_L$$

Then, it is easy to see that every model $M_\rho$ that satisfies $\Gamma$ must hold $\rho(p) \in (\bigcap_{i \in I} V_i)$, and then $(\bigcap_{i \in I} V_i) \subseteq V$. Therefore, we can assure that if we apply repeatedly the *composition* and *weakening* rules, we also can deduce syntactically $(p, V)$. Then the theorem is true for $n = 0$.

**Induction hypothesis:** The theorem is true for $n - 1$.

**Case** $n$**:** Suppose that $p \in S_n$. Given $R_p$, the set of rules of $\Gamma$ with conclusion $p$, then $\forall r \in R_p$, the premises of $r$ belong to lower layers. Let be $V_q = \bigcap\{V | \Gamma \models (q, V)\}$, $\forall q \in Prem(r)$, $\forall r \in R_p$. By the induction hypothesis we have that $\Gamma \models (q, V_q) \Rightarrow \Gamma \vdash (q, V_q)$, $\forall q \in Prem(r)$, $\forall r \in R_p$.

By induction over $nrp$, the number of rules of $R_p$, and together with the conditions of the theorem, we will prove that $\Gamma \models (p, V)$ implies $\Gamma \vdash (p, V)$.

1. $nrp = 1$. In this case we have $R_p = \{(q_1 \wedge \cdots \wedge q_m \rightarrow p, W)\}$, $\Gamma \models (q_i, V_{q_i})$ for $i = 1, \ldots, m$, where $V_{q_i}$ are minimals. From Lemma 2 we have that $\Gamma \models (p, V)$ if and only if

$$\{(q_1 \wedge \cdots \wedge q_m \rightarrow p, W)(q_1, V_{q_1}), \ldots, (q_m, V_{q_m})\} \models (p, V)$$

but from properties SR-4 and SR-5 of proposition 2, this holds if and only if

$$V \supseteq MP_T^*(T^*(V_{q_1}, V_{q_2}, \ldots, V_{q_m}), W)$$

Given that

$$MP_T^*(T^*(V_{q_1}, V_{q_2}, \ldots, V_{q_m}, W) = MP_T^*(V_{q_1}, MP_T^*(V_{q_2}, \ldots, MP_T^*(V_{q_m}, W) \cdots))$$

it is easy to see that by successive applications of SIR inference rule we can obtain $\Gamma \vdash (p, V)$, and thus for $nrp = 1$ the theorem is true.

2. Suppose that the theorem is true for $nrp = k - 1$.

3. $nrp = k$. In this case we have that $R_p = \{r_1, r_2, \ldots, r_k\}$ and $\Gamma \models (q_{ij}, W_{q_{ij}})$ for all $q_{ij} \in Prem(r_i)$, $i = 1, \ldots, k$, being $W_{q_{ij}}$ minimal. Let be $r_i = (\wedge q_{ij} \rightarrow p, V_i)$ and $A_p = \bigcup_{i,j}(q_{ij}, W_{q_{ij}})$. Again Lemma 2 allows us to state that $\Gamma \models (p, V)$ if and only if $V \supseteq U$, where $\{r_1, \ldots, r_k\} \cup A_p \models (p, U)$ and $U$ minimal,. Consider $R_p = R_p^* \cup r_k$, where $R_p^* = \{r_1, \ldots, r_{k-1}\}$ and let be $V^* = \bigcap\{V'' | \{R_p^*, A_p\} \models (p, V'')\}$. By induction hypothesis we have also $R_p^* \vdash (p, V^*)$. Furthermore we have $\{r_k, A_p\} \vdash (p, MP_T^*(T^*(W_{q_{k1}}, \ldots, W_{q_{kj_k}}, V_k))$, and because of Lemma 1 we know that $MP_T^*(T^*(W_{q_{k1}}, \ldots, W_{q_{kj_k}}, V_k) = \bigcap\{V'' | \{r_k, A_p\} \models (p, V'')\}$. Then from Lemma 1, we have:

$$\{R_p, A_p\} \models (p, V) \text{ iff } V \supseteq V^* \cap MP_T^*(T^*(W_{q_{k1}}, \ldots, W_{q_{kj_k}}, V_k)$$

Finally it is easy to notice that $V^* \cap MP_T^*(T^*(W_{q_{k1}}, \ldots, W_{q_{kj_k}}, V_k)$ can be obtained by successive applications of SIR and *composition* inference rules, that is, we can finally conclude that $\Gamma \vdash (p, V)$.□

# Appendix C

# Code Examples

## C.1 *Terap–IA* Example

In the following we present an small selection of the code of *Terap–IA* that has been explained in Section 6.2.

```
;; 17-3-93
;; MODULS DE TRACTAMENT DE LES PNEUMONIES ATIPIQUES

Module pneumonia_chlam_pneum_tractament_1: antibiotics_chlamydia_pneum =
 Begin
 Inherit ant
 Inherit situacio_clinica
 Open insuf_renal:
  Begin
  Export  quinol, tetras_1, tetras_2, macrol
  End
 Export  cipro, oflox, tetras_ac_rap, doxi, doxi_DI,
        eritro_DB, eritro_DA, roxi
 Deductive Knowledge
 Rules:
     ;; quinolones per chlamydia_pneum
   R004 If quinol then conclude cipro is modp
   R005 If quinol and situacio_clinica/tract_OR then conclude oflox is p
     ;;tetraciclines per chlamydia_pneum
   R006 If tetras_1 then conclude tetras_ac_rap is mp
   R007 If tetras_2 then conclude doxi is mp
   R008 If tetras_2 then conclude doxi_DI is mp
     ;; macrolids per chlamydia_pneum
   R001 If macrol then conclude eritro_DB is fp
   R002 If macrol then conclude eritro_DA is fp
   R003 If macrol and situacio_clinica/tract_OR then conclude roxi is p
 End deductive
 End
;;-------------------------------
;; Atmar RL, Greenberg SB, Pneumonia caused by Mycoplasma
;; pneumoniae and the TWAR Agent. Semin Respir Infect 1989; 4:
;; 19-31
```

```
;; Grayston JT, Wang SP, Kuo CC,Campbell LA, Current  knowledge
;; on Chlamydia Pneumoniae , strain TWAR, an important cause of
;; pneumonia and other acute respiratory diseases. Eur J Clin
;; Microbiol Infect Dis 1989; 8: 191-202

;; Lipski BA, Tack KJ, Kuo C, Wang S, Grayston T. Ofloxacin
;; treatment of Chlamydia Pneumoniae (Strain TWAR) Lower
;; Respiratory Tract Infections. Am J Med 1990,89: 722-724

;; Grayston JT, Thom DH. The chlamydial pneumonias, in Remington
;; JS, Swartz MN. Current Clinical Topics in Infectious Diseases.
;;  Boston, Blackwell Scientific Publications, 1991: 1-18

;; Finegold SM. Aspiration Pneumonia, Lung abscess and Empiema.
;; Pennington JE. Respiratory Infections: Diagnosis and Management.
;; Raven Press, New York 1988: 264_275
;;--------------------------------

Module antibiotics_chlamydia_pneum=antimicrobians:
   Begin
  Export cipro, oflox, tetras_ac_rap, doxi, doxi_DI, eritro_DB,
        eritro_DA, roxi
   End


Module ABS=
;; MODUL ABS GRUPS D'ANTIMICROBIANS QUE S'UTILITCEN A TERAPIA
   Begin
  Export  quinol, tetras_1, tetras_2, cotri, sulfas, vanco, teico, amino,
         metro, clinda, carbapen, INH, RFM, ETM, PZ, anf_B, ACV, GCV,
         ARA_A, RBV, AMD, RMD, peni, macrol, b_lactam_inh, cef, monobac
  Deductive knowledge
  Dictionary:
 Predicates:
   quinol= name: "quinolones"
          type: logic
tetras_1= name: "tetraciclines d'accio rapida"
         type: logic
         relation: belongs_to_group tetraciclines
tetras_2= name: "tetraciclines de accio retardada"
         type: logic
         relation: belongs_to_group tetraciclines
cotri= name: "cotrimoxazol"
      type: logic
sulfas= name: "sulfamidas"
       type: logic
vanco= name: "vancomicina"
      type: logic
teico= name: "teicoplanina"
      type: logic
amino= name: "aminoglucosids"
      type: logic
metro= name: "metronidazol"
      type: logic
clinda= name: "clindamicina"
      type: logic
```

```
carbapen= name: "carbapenems"
         type: logic
         relation: belongs_to_group atb_betalactamics
INH= name: "isoniacida"
    type: logic
    relation: belongs_to_group tuberculostatics
RFM= name: "rifampicina"
    type: logic
    relation: belongs_to_group tuberculostatics
ETM= name: "etambutol"
    type: logic
    relation: belongs_to_group tuberculostatics
PZ= name: "pirazinamida"
   type: logic
   relation: belongs_to_group tuberculostatics
anf_B= name: "anfotericina_B"
      type: logic
ACV= name: "aciclovir"
    type: logic
    relation: belongs_to_group antivirics
GCV= name: "ganciclovir"
    type: logic
    relation: belongs_to_group antivirics
ARA_A= name: "vidarabina"
      type: logic
      relation: belongs_to_group antivirics
RBV= name: "ribaravina"
    type: logic
    relation: belongs_to_group antivirics
AMD= name: "amantadina"
    type: logic
    relation: belongs_to_group antivirics
RMD= name: "rimantadina"
    type: logic
     relation: belongs_to_group antivirics
peni= name: "penicil.lines"
     type: logic
     relation: belongs_to_group atb_betalactamics
macrol= name: "macrolids"
      type: logic
b_lactam_inh= name: "inhibidors de les betalactamases"
             type: logic
             relation: belongs_to_group atb_betalactamics
cef= name: "cefalosporines"
    type: logic
    relation: belongs_to_group atb_betalactamics
monobac= name: "monobactams"
        type: logic
        relation: belongs_to_group atb_betalactamics
atb_betalactamics= name: "antibiotics betalactamics"
                 type:class
tetraciclines name: "tetraciclines"
             type: class
antivirics= name: "farmacs antivirics"
           type: class
tuberculostatics= name: "agents antituberculosos"
                type: class
```

```
 End deductive
  End

;; 18-3-93
;;----------------------DOCUMENT CONDICIONS GENERALS-------------
;; EL MODUL ABS_1 ES UN REFINAMENT DEL MODUL ABS (VEURE
;; DOCUMENT DIC) EN EL QUE ES DONA UN VALOR DE CERTESA SEGUR A
;; TOTS ELS GRUPS D'ANTIBIOTICS UTILITZATS A TERAP-IA

Module ABS_1:ABS =
  Begin
  Export  quinol, tetras_1, tetras_2, cotri, sulfas, vanco, teico, amino,
         metro, clinda, carbapen, INH, RFM, ETM, PZ, anf_B, ACV, GCV,
         ARA_A, RBV, AMD, RMD, peni, macrol, b_lactam_inh, cef, monobac
  Deductive Knowledge
  Rules:
        R001 If s then conclude quinol is s
        R002 If s then conclude tetras_1 is s
        R003 If s then conclude tetras_2 is s
        R004 If s then conclude cotri is s
        R005 If s then conclude sulfas is s
        R006 If s then conclude vanco is s
        R007 If s then conclude teico is s
        R008 If s then conclude amino is s
        R009 If s then conclude metro is s
        R010 If s then conclude clinda is s
        R011 If s then conclude carbapen is s
        R012 If s then conclude INH is s
        R013 If s then conclude RFM is s
        R014 If s then conclude ETM is s
        R015 If s then conclude PZ is s
        R016 If s then conclude anf_B is s
        R017 If s then conclude ACV is s
        R018 If s then conclude GCV is s
        R019 If s then conclude ARA_A is s
        R020 If s then conclude RBV is s
        R021 If s then conclude AMD is s
R022 If s then conclude RMD is s
        R023 If s then conclude peni is s
        R024 If s then conclude macrol is s
        R025 If s then conclude b_lactam_inh is s
        R026 If s then conclude cef is s
        R027 If s then conclude monobac is s
        R028 If s then conclude (macrol plus RFM) is s
        R029 If s then conclude (peni plus metro) is s
        R030 If s then conclude (macrol plus metro) is s
  End deductive
   End

;; ELS MODULS GESTACIO, LACTANCIA, ALERGIA, INSUFICIENCIA RENAL I
;; FACTORS GENETICS SON MODULS DE CONDICIONS GENERALS DEL
;; PACIENT QUE MODIFIQUEN LA TERAPIA I QUE ES PODEN APLICAR SOBRE
;; GRUPS D'ANTIBIOTICS. CADA UNA D'AQUESTES CONDICIONS  AFECTEN
;; PER IGUAL A TOTS ELS ANTIBIOTIC DE UN MATEIX GRUP I PER TANT ES
;; PODEN APLICAR SOBRE EL GRUP

Module insuf_renal:ABS=
```

```
   Begin
  Module x = alergia
    ;; anam_R es un refinament d'anam per insuficiencia renal
  Module anam_R= anam:
     Begin
    Export insuf_renal
     End
  Export  quinol, tetras_1, tetras_2, cotri, sulfas, vanco, teico, amino,
         metro, clinda, carbapen, INH, RFM, ETM, PZ, anf_B, ACV, GCV,
         ARA_A, RBV, AMD, RMD, peni, macrol, b_lactam_inh, cef, monobac
  Deductive Knowledge
  Rules:
     R001 If anam_R/insuf_renal then conclude no(tetras_1) is s
     R002 If anam_R/insuf_renal then conclude tetras_2 is p
     R003 If anam_R/insuf_renal then conclude amino is p
     R004 If anam_R/insuf_renal then conclude anf_B is p
     R005 If anam_R/insuf_renal then conclude AMD is p
  End Deductive
  Control knowledge
  Evaluation type: eager
  Deductive control:
     M001 If K(x/$y,$c) and NP($y) then conclude K($y,$c)
  End control
   End

Module Alergia:ABS=
   Begin
  Module x = lactancia
   ;; anam_R es anam refinat per alergia
  Module anam_R= anam:
    Begin
   Export alergia, alergia_grups_ABS, alergia_inmed_peni,
             alergia_retard_peni, alergia_infrec
    End
  Export  quinol, tetras_1, tetras_2, cotri, sulfas, vanco, teico, amino,
         metro, clinda, carbapen, INH, RFM, ETM, PZ, anf_B, ACV, GCV,
         ARA_A, RBV, AMD, RMD, peni, macrol, b_lactam_inh, cef, monobac
  Deductive Knowledge
  Rules:
        R001 If anam_R/alergia_grups_ABS=(quinol) then conclude no(quinol) is s
        R002 If anam_R/alergia_grups_ABS=(tetras_1)
                then conclude no(tetras_1) is s
        R003 If anam_R/alergia_grups_ABS=(tetras_2)
                then conclude no(tetras_2) is s
        R004 If anam_R/alergia_grups_ABS=(cotri) then conclude no(cotri) is s
        R005 If anam_R/alergia_grups_ABS=(sulfas) then conclude no(cotri) is s
        R006 If anam_R/alergia_grups_ABS=(sulfas) then conclude no(sulfas) is s
        R007 If anam_R/alergia_grups_ABS=(vanco) then conclude no(vanco) is s
        R008 If anam_R/alergia_grups_ABS=(teico) then conclude no(teico) is s
        R009 If anam_R/alergia_grups_ABS=(vanco) then conclude teico is llp
        R010 If anam_R/alergia_grups_ABS=(amino) then conclude no(amino) is s
        R011 If anam_R/alergia_grups_ABS=(metro) then conclude no(metro) is s
        R012 If anam_R/alergia_grups_ABS=(clinda) then conclude no(clinda) is s
        R013 If anam_R/alergia_grups_ABS=(carbapen) then conclude no(carbapen) is s
        R014 If anam_R/alergia_inmed_peni then conclude no(carbapen) is s
        R015 If anam_R/alergia_retard_peni then conclude carbapen is p
        R016 If anam_R/alergia_infrec then conclude carbapen is p
```

```
        R017 If anam_R/alergia_grups_ABS=(INH) then conclude no(INH) is s
        R018 If anam_R/alergia_grups_ABS=(RFM) then conclude no(RFM) is s
        R019 If anam_R/alergia_grups_ABS=(ETM) then conclude no(ETM) is s
        R020 If anam_R/alergia_grups_ABS=(PZ) then conclude no(PZ) is s
        R021 If anam_R/alergia_grups_ABS=(anf_B) then conclude no(anf_B) is s
        R022 If anam_R/alergia_grups_ABS=(ACV) then conclude no(ACV) is s
        R023 If anam_R/alergia_grups_ABS=(GCV) then conclude no(GCV) is s
        R024 If anam_R/alergia_grups_ABS=(ARA_A) then conclude no(ARA_A) is s
        R025 If anam_R/alergia_grups_ABS=(RBV) then conclude no(RBV) is s
        R026 If anam_R/alergia_grups_ABS=(AMD) then conclude no(AMD) is s
        R027 If anam_R/alergia_grups_ABS=(RMD) then conclude no(RMD) is s
        R028 If anam_R/alergia_grups_ABS=(peni) then conclude no(peni) is s
        R029 If anam_R/alergia_grups_ABS=(macrol) then conclude no(macrol) is s
        R030 If anam_R/alergia_grups_ABS=(b_lactam_inh)
                then conclude no(b_lactam_inh) is s
        R031 If anam_R/alergia_grups_ABS=(peni)
                then conclude no(b_lactam_inh) is s
        R032 If anam_R/alergia_grups_ABS=(cef) then conclude no(cef) is s
        R033 If anam_R/alergia_inmed_peni then conclude no(cef) is s
        R034 If anam_R/alergia_retard_peni then conclude cef is p
        R035 If anam_R/alergia_infrec then conclude cef is p
        R036 If anam_R/alergia_grups_ABS=(monobac)
               then conclude no(monobac) is s
        R037 If anam_R/alergia_grups_ABS=(metro)
                then conclude no(peni plus metro) is s
        R038 If anam_R/alergia_grups_ABS=(peni)
                then conclude no(peni plus metro) is s
        R039 If anam_R/alergia_grups_ABS=(macrol)
                then conclude no(macrol plus metro) is s
        R040 If anam_R/alergia_grups_ABS=(metro)
                then conclude no(macrol plus metro) is s
        R041 If anam_R/alergia_grups_ABS=(RFM)
                then conclude no(macrol plus RFM) is s
        R042 If anam_R/alergia_grups_ABS=(macrol)
                then conclude no(macrol plus RFM) is s
  End Deductive
  Control knowledge
     Evaluation type: eager
     Deductive control:
      M001 If K(x/$y,$c) and NP($y) then conclude K($y,$c)
  End control
   End

;; LACTANCIA  NO MODIFICA EL VALOR DELS GRUPS
;; D'ANTIBIOTICS. SI UN ANTIBIOTIC NO ESTA RECOMANAT EN LA
;; LACTANCIA S'ACONSELLA DEIXAR DE LACTAR.

Module lactancia:ABS=
;; falta posar bibliografia
   Begin
  Module X = gestacio
  Module AD = anam_dona
  Export  quinol, tetras_1, tetras_2, cotri, sulfas, vanco, teico, amino,
         metro, clinda, carbapen, INH, RFM, ETM, PZ, anf_B, ACV, GCV,
         ARA_A, RBV, AMD, RMD, peni, macrol, b_lactam_inh, cef, monobac
  Deductive knowledge
  Dictionary:
```

```
 Predicates:
     d_d_lact= name: "deixar de lactar"
             type: logic
  Rules:
   R001 If quinol and AD/lact  then conclude d_d_lact is s
   R002 If cotri and AD/neonatg6pd  then conclude d_d_lact is s
   R003 If cotri and AD/premat  then conclude d_d_lact is s
   R004 If teico and AD/lact  then conclude d_d_lact is s
   R005 If sulfas and AD/neonatg6pd  then conclude d_d_lact is s
   R006 If sulfas and AD/premat  then conclude d_d_lact is s
   R007 If metro and AD/lact  then conclude d_d_lact is s
   R008 If GCV and AD/lact   then conclude d_d_lact is s
   R009 If ARA_A and AD/lact  then conclude d_d_lact is s
   R010 If RBV and AD/lact   then conclude d_d_lact is s
   R011 If (peni plus metro) and AD/lact  then conclude d_d_lact is s
   R012 If (macrol plus metro) and AD/lact then conclude d_d_lact is s
  End Deductive
  Control knowledge
    Evaluation type: eager
    Deductive control:
       M001 If K(x/$y,$c) and NP($y) then conclude K($y,$c)
   End control
    End

Module Gestacio:ABS=
   Begin
  Module x = ABS_1
  Module AD =anam_dona
  Export  quinol, tetras_1, tetras_2, cotri, sulfas, vanco, teico, amino,
         metro, clinda, carbapen, INH, RFM, ETM, PZ, anf_B, ACV, GCV,
         ARA_A, RBV, AMD, RMD, peni, macrol, b_lactam_inh, cef, monobac
  Deductive Knowledge
  Dictionary:
 Predicates:
   adm_GCV_si_no_alt= name: "administrar el farmac nomes si no hi ha un
                             altra alternativa i es imprescindible"
                      type: logic
   adm_RBV_si_no_alt= name: "administrar el farmac  nomes si no hi ha un
                             altra alternativa i es imprescindible"
                      type: logic
  Rules:
    R001 If AD/gest then conclude no(quinol) is s
    R002 If AD/gest then conclude no(tetras_1) is s
    R003 If AD/gest then conclude no(tetras_2) is s
    R004 If AD/gest_t  then conclude no(cotri) is s
    R005 If AD/gest  then conclude cotri is p
    R006 If AD/gest_t then conclude no(sulfas) is s
    R007 If AD/gest then conclude sulfas is p
    R008 If AD/gest then conclude vanco is p
    R009 If AD/gest_t then conclude no(teico) is s
    R010 If AD/gest  then conclude amino is p
    R011 If AD/temps_gest=(gest_1_t) then conclude no(metro) is s
    R012 If AD/temps_gest=(gest_2_t or gest_3_t) then conclude metro is llp
    R013 If AD/gest then conclude clinda is p
    R014 If AD/gest then conclude carbapen is p
    R015 If AD/gest then conclude RFM is p
    R016 If AD/gest then conclude PZ is llp
```

```
    R017 If AD/gest then conclude anf_B is p
         R018 If AD/gest then conclude ACV is p
    R019 If AD/gest then conclude GCV is llp
    R020 If AD/gest then conclude no(ARA_A) is s
    R021 If AD/gest then conclude RBV is llp
    R022 If AD/gest then conclude no(AMD) is s
    R023 If AD/gest then conclude RMD is llp
    R024 If AD/gest then conclude (macrol plus RFM) is p
    R025 If AD/temps_gest=(gest_1_t)
                           then conclude no(peni plus metro) is s
    R026 If AD/temps_gest=(gest_2_t or gest_3_t)
                           then conclude (peni plus metro) is llp
   End Deductive
  Control knowledge
     Evaluation type: eager
     Deductive control:
       M001 If K(x/$y,$c) and NP($y) then conclude K($y,$c)
  End control
   End

;;18-3-93
;;------------------------------MODULS MENU-------------------------

;; ELS MODULS QUE ANOMENO "MENU" SON MODULS DE PREGUNTES AL
;; USUARI DE LES DADES DEL MALALT QUE AJUDEN A DECIDIR EL
;; TRACTAMENT DE LES PNEUMONIES  INCLOU EL MODUL DIAG QUE
;; PREGUNTA AL USUARI QUINS GERMENS VOL TRACTAR

MODULE DIAG=
;; DIAG PREGUNTA AL USUARI QUINS GERMENS VOL TRACTAR
;; pendent demanar la certesa del diag de un altra forma
  Begin
  Import diagnostics, pneum_leg, pneum_asper, pneum_tbc
  Export diagnostics, pneum_leg, pneum_asper, pneum_tbc
  Deductive Knowledge
  Dictionary:
 Predicates:
   diagnostics=
    name: "diagnostics"
    question: "quina de les seguents vol tractar"
    type: (pneum_myc or pneum_cox or pneum_chlam_psit or pneum_chlam_pneum
         or pneum_leg or pneum_pneum or pneum_anaer or pneum_enterobac or
         pneum_H_inf or pneum_branh or pneum_pseud or pneum_meningo or
         pneum_S_pyog or pneum_S_aur or pneum_asper or pneum_crip
         or pneum_nocar or pneum_CMV or pneum_VVZ or pneum_HSV or pneum_EBV
         or pneum_VRS or pneum_ADV or pneum_influenza)
   pneum_leg= name: "pneumonia per legionella pneumophila"
             question: "quina es la certesa de pneum_leg"
             type: logic
   pneum_asper= name: "pneumonia per aspergilus"
               question: "quina es la certesa de pneum_aspergilus"
               type: logic
   pneum_tbc= name: "pneumonia tuberculosa"
             question: "quina es la certesa de pneum_myc_tuberculosis"
             type: logic
 End deductive
End
```

```
MODULE ANAM_GENERAL=
;; ES UN MODUL EN EL QUE ES DEFINEIXEN LES INTERFICIES D'IMPORTACIO
;; I EXPORTACIO DE LES DADES  MES GENERALS DE L'ANAMNESI
   Begin
    Export edat, sexe, alergia, alergia_grups_ABS, alergia_inmed_peni,
            alergia_retard_peni, alergia_infrec, reaccions_adv_atb,
            g6pd, insuf_hepatica, insuf_renal
   End

MODULE ANAM_SPEC:ANAM_GENERAL=
;; ES UN MODUL DE REFINAMENT D'ANAM GENERAL EN EL QUE ES
;; DEFINEIXEN ELS SUBMODULS ANT I ANALITICA QUE ES NECESITAN PER
;; DEDUIR EL VALOR DEL FET INSUFICIENCIA RENAL.I ES DECLAREN ELS FETS
;; EN EL DICCIONARI
   Begin
     Export edat, sexe, alergia, alergia_grups_ABS, alergia_inmed_peni,
      alergia_retard_peni, alergia_infrec,reaccions_adv_atb, g6pd,
      insuf_hepatica, insuf_renal
  Deductive Knowledge
 Dictionary:
 Predicates:
   edat= name: "edat"
        question: "quina es l'edat del pacient?"
        type: numeric
   sexe= name: "sexe"
        question:"es una dona o un home?"
        type: (home or dona)
   insuf_renal= name: "insuficiencia renal"
               type: logic
   alergia= name:"alergia a antibiotics"
           question: "hi han antecedents d'alergia a antibiotics?"
           type: boolean
   alergia_grups_ABS=
           name:"alergia a algun grup d' antibiotics "
           question: "hi han antecedents d' alergia a algun dels seguents
                        grups d' antibiotics?"
           type: (peni or cef or monobac or carbapen or b_lactam_inh or
                     tetras_1 or tetras_2 or quinol or amino or clinda or cotri
                     or INH or RFM or ETM or PZ or macrol or vanco or anf_B or
                     ACV or GCV or ARA_A or RBV or AMD or RMD or metro or sulfas
                     or teico)
           relation: needs alergia
    alergia_inmed_peni=
           name:"reaccio alergica inmediata a la penicil.lina "
           question: "la reaccion alergica a la penicil.lina s'ha
                      produit en les primeres 72 hores de l'administracio
                      del antibiotic?"
           type: boolean
           relation: needs alergia_grups_ABS
   alergia_retard_peni=
           name:"reaccio alergica retardada a la penicil.lina"
           question: "la reaccion alergica s'ha produit despres
                      de 72 hores de l'administracio del antibiotic?"
           type: boolean
           relation: needs alergia_grups_ABS
           relation: needs alergia_inmed_peni
```

```
   alergia_infrec=
           name:"reaccions alergiques infrecuents"
           question: "hi han antecedents de anemia hemolitica,
                     infiltrats pulmonars amb eosinofilia, nefritis
                     intersticial, granulopenia, trombocitopenia,
                     febre per drogues, vasculitis per hiper
                     sensibilitat, eritema multiforme, sindrom lupus like?"
           type: boolean
           relation: needs alergia_grups_ABS
           relation: needs alergia_inmed_peni
           relation: needs alergia_retard_peni
   reaccions_adv_atb=
name:"reaccions adverses a antibiotics"
question: "ha presentat reaccions adverses a algun
antibiotic?"
type: boolean
   g6pd=  name: "deficit de glucosa 6 fosfat deshidrogenasa"
         question:"Te el pacient un deficit de glucosa 6 fosfat
                      deshidrogenasa?"
         type: boolean
   insuf_hepatica= name: "insuf_hepatica"
                  question: "hi han signes d'insuficiencia hepatica?"
                  type: boolean
    End deductive
    End

MODULE ANAM:ANAM_SPEC=
;; ANAM ES UN REFINAMENT D'ANAM_SPEC. EN AQUEST MODUL HI HAN LES
;; REGLES QUE PERMETEN DEFINIR EL FET INSUFICIENCIA RENAL I LES
;; METARREGLES  QUE ORDENAN DE FORMA LLOGICA LES PREGUNTES SOBRE
;; ANTECEDENTS D'ALERGIA
   Begin
   Inherit ant
   Inherit analitica
   Import edat, sexe, alergia, alergia_grups_ABS, alergia_inmed_peni,
           alergia_retard_peni, alergia_infrec, reaccions_adv_atb, g6pd,
           insuf_hepatica
   Export edat, sexe, alergia, alergia_grups_ABS, alergia_inmed_peni,
           alergia_retard_peni, alergia_infrec, reaccions_adv_atb, g6pd,
           insuf_hepatica, insuf_renal
   Deductive Knowledge
   Rules:
     R001 If analitica/insuf_renal_ag then conclude insuf_renal is s
     R002 If ant/mal_cron_assoc and
               ant/tipus_mal_cron_assoc=(insuf_renal_cron)
               then conclude insuf_renal is s
   End deductive
   Control Knowledge
   Deductive control:
    M001 If K(not(alergia),s) then conclude K(=(alergia_grups_ABS, none), s)
    M002 If K(not(alergia_grups_ABS),s)
              then conclude K(not(alergia_inmed_peni),s)
    M003 If K(not(alergia_grups_ABS),s)
              then conclude K(not(alergia_retard_peni),s)
    M004 If K(not(alergia_grups_ABS),s)
              then conclude K( not (alergia_infrec), s)
     M005 If K(=(alergia_grups_ABS,$x),s) and no(member($x,(peni)))
```

```
              then conclude K(not(alergia_inmed_peni),s)
    M006 If K(=(alergia_grups_ABS,$x),s) and no(member($x,(peni)))
              then conclude K(not(alergia_retard_peni),s)
    M007 If K(=(alergia_grups_ABS,$x),s) and no(member($x,(peni)))
              then conclude K( not (alergia_infrec), s)
    M008 If K(alergia_inmed_peni, s)
              then conclude K(not(alergia_retard_peni),s)
    M009 If K(alergia_inmed_peni, s)
              then conclude K( not (alergia_infrec), s)
    M010 If K(alergia_retard_peni, s)
              then conclude K( not (alergia_infrec), s)
   End control
   End

MODULE ANAM_DONA=
;; ANAM_DONA PREGUNTA AL USUARI DADES ESPECIFIQUES PER DONES
   Begin
   Open anam
   Import gest, temps_gest, gest_t, lact, premat, neonatg6pd
   Export gest, temps_gest, gest_t, lact, premat, neonatg6pd, sexe, edat
   Deductive Knowledge
   Dictionary:
  Predicates:
    gest= name: "gestacio"
         question: "es una gestant?"
         type: boolean
          relation: needs sexe
         relation: needs edat
    temps_gest= name: "temps de gestacio"
               question: "esta en el: primer trimestre de la
                             gestacio(gest_1_t) segon trimestre de la
                             gestacio(gest_2_t) tercer trimestre de la
                             gestacio(gest_3_t)?"
              type: (gest_1_t or gest_2_t or gest_3_t )
              relation: needs gest
    gest_t= name: "gestacio a terme"
           question: "es una gestant a terme?"
           type: boolean
           relation: needs gest
           relation: needs temps_gest
    lact= name:"lactancia"
         question: "esta en periode de lactancia?"
         type: boolean
         relation: needs sexe
         relation: needs edat
         relation: needs gest
    neonatg6pd= name: "neonat amb deficit de glucosa 6 fosfat deshidrogenasa"
               question:"Te el lactant un deficit de glucosa 6 fosfat
                            deshidrogenasa?"
               type: boolean
               relation: needs lact
    premat= name: "prematur"
           question: "es el lactant prematur?"
           type: boolean
           relation: needs lact
   End deductive
   Control knowledge
```

```
    Deductive control:
    M001 If K(gest,s) then conclude K(not(lact),s)
    M002 If K(=(sexe, (home) ),s) then conclude K(not(gest),s)
    M003 If K(=(sexe, (home) ),s) then conclude K(not(lact),s)
    M004 If K(=(sexe, (dona) ),s) and K(=(edat,$x),s) and lt($x,15)
              then conclude K(not(gest),s)
    M005 If K(=(sexe, (dona) ),s) and K(=(edat,$x),s) and lt($x,15)
              then conclude K(not(lact),s)
    M006 If K(=(sexe, (dona) ),s) and K(=(edat,$x),s) and gt($x,45)
              then conclude K(not(gest),s)
    M007 If K(=(sexe, (dona) ),s) and K(=(edat,$x),s) and gt($x,45)
              then conclude K(not(lact),s)
    M008 If K(not(gest),s) then conclude K(not(gest_t),s)
    M009 If K(not(gest),s) then conclude K(=(temps_gest, none ),s)
    M010 If K(not(lact),s) then conclude K(not(premat),s)
    M011 If K(not(lact),s) then conclude K(not(neonatg6pd),s)
    M012 If K(=(temps_gest,$x),s) and member ($x, (gest_1_t or gest_2_t))
            then conclude K( not (gest_T),s)
   End control
   End

MODULE ANT=
;; EL MODUL ANT PREGUNTA ELS ANTECEDENTS PATOLOGICS D'INTERES
  Begin
   Import mal_cron_assoc, tipus_mal_cron_assoc, inmuno, tipus_inmuno,
           atb_betalactamics_previs, tract_assoc, hosp_previa, pneum_previa
   Export mal_cron_assoc, tipus_mal_cron_assoc, inmuno, tipus_inmuno,
           atb_betalactamics_previs, tract_assoc, hosp_previa, pneum_previa
   Deductive Knowledge
   Dictionary:
  Predicates:
    mal_cron_assoc= name: "malaltia cronica associada"
                   question: "hi han antecedents de malalties croniques
                             associades?"
                   type:boolean
    tipus_mal_cron_assoc=
         name:"tipus de malaltia cronica associada"
         question: "hi han antecedents de:
                            Malaltia hepatica cronica (hepat_cron)
                            Insuficiencia cardiaca avanzada (IC)
                            Diabetes mellitus (DB)
                            Alcoholisme (OH)
                            Malaltia pulmonar obstructiva cronica (EPOC)
                             Vasculitis o colagenosis (vasc_colag)
                            Sarcoidosis (sarc)
                            Drogadiccio parenteral (ADVP)
                            Neoplasia avanzada (neop )
                            Insuficiencia renal cronica (insuf_renal_cron)"
         type: (hepat_cron or IC or DB or OH or EPOC or vasc_colag or sarc
                  or ADVP or neop or insuf_renal_cron)
          relation: needs mal_cron_assoc
    tipus_inmuno=
         name: "tipus d'inmunosupresio"
        question: "hi han antecedents de:
                 tractament amb corticoides > 5 mgrs al dia o
                 drogues citotoxiques en els ultims sis mesos (tract_inmuno)
                 transplant de medula osea (TMO)
```

```
                 transplant d'altres organs (TAO)
                 infeccio HIV (HIV)
                  hipogamaglobulinemia o agammaglobulinemia (alt_IG)"
        type: (tract_inmuno or TMO or TAO or HIV or alt_IG)
         relation: needs inmuno
    inmuno= name: "inmunosupresio"
           question: "hi han antecedents d'inmunosupresio?"
           type: boolean
           relation: needs mal_cron_assoc
    hosp_previa=  name:" hospitalitzacio previa"
                 question: " el pacient ha estat hospitalitzat:
                            en el ultim any (hosp_1_any)
                            en els ultims tres mesos (hosp_3_m)
                            no ha estat hospitalitzat recentment (no_hosp)?"
                type: ( hosp_1_any or hosp_3_m or no_hosp)
    pneum_previa= name:" pneumonia previa"
                 question: " el pacient ha sofert una pneumonia
                            durant l'ultim any?"
                 type: boolean
    atb_betalactamics_previs=
                name: "antibiotics betalactamics previs"
             question: "hi han antecedents de us de antibiotics
                       betalactamics en els ultims tres mesos?"
             type: boolean
    tract_assoc=
         name: "tractaments associats"
         question: "el pacient pren habitualment algun dels seguents farmacs:
                   teofilina (teof)
                   carbamacepina (carbam)
                   digoxina (digox)
                   dicumarinics (dicum)
                   ciclosporina (ciclos)
                   difenilhidantoina (DFH)?"
        type: (teof or carbam or digox or dicum or ciclos or DFH)
   End deductive
   Control knowledge
    Deductive control:
    M001 If K(not(mal_cron_assoc),s)
             then conclude K(=(tipus_mal_cron_assoc, none ),s)
    M002 If K(not(mal_cron_assoc),s) then conclude K(not(inmuno),s)
    M003 If K(not(inmuno),s) then conclude K(=(tipus_inmuno, none ),s)
   End control
   End

MODULE SITUACIO_CLINICA=
;; MODUL PER EVALUAR L'ESTAT CLINIC DEL PACIENT AL INGRES O
;; VISITA
  Begin
 Import estat_malalt, febre, TAs, TAd, FC, FR, alt_GI, trans_degl,
         shock_septic, alt_neurol
 Export estat_malalt, febre, FR, alt_GI, trans_degl, shock_septic, alt_neurol,
         signes_clin_grav, tract_OR, tract_parenteral
 Deductive knowledge
 Dictionary:
Predicates:
   alt_GI= name:"alteracions gastrointestinals"
          question: "te basques o vomits que dificulten l'ingesta oral?"
```

```
       type: boolean
       relation: needs estat_malalt
 trans_degl= name: "transtorns en la deglucio"
           question: "te dificultats per empassar?"
           type:boolean
           relation: needs estat_malalt
 febre= name:"febre"
      question: "quina es la temperatura?"
      type: numeric
 TAs= name:"tensio arterial sistolica "
     question: "quina es la tensio arterial sistolica?"
     type: numeric
     relation: needs TAd
 TAd= name:"tensio arterial diastolica "
     question: "quina es la tensio arterial diastolica?"
     type: numeric
 FC= name:"frecuencia cardiaca"
    question: "quina es la frecuencia cardiaca?"
    type: numeric
  FR= name:"frecuencia respiratoria "
    question: "quina es la frecuencia respiratoria?"
    type: numeric
 shock_septic= name: "shock septic"
             question:"la TAs es<90mHg i s'observan signes de
                      hipoperfusio periferica?"
             type: boolean
             relation: needs estat_malalt
             relation: needs TAs
 alt_neurol= name: "alteracions neurologiques"
           question: " Hi ha obnubilacio o coma?"
           type: boolean
           relation: needs estat_malalt
 estat_malalt= name: "estat del malalt"
             question: "quin es segons voste l'estat_malalt del malalt:
                       lleu
                       moderadament greu (mod_g)
                       greu
                       molt greu (molt_g) ?"
             type: ( lleu or mod_g or greu or molt_g)
 signes_clin_grav= name:"signes clinics de gravetat"
                 type: logic
 tract_parenteral= name: "tractament parenteral"
                 type: logic
 tract_OR= name: "tractament oral"
         type: logic
Rules:
 R001 If TAs<90 then conclude signes_clin_grav is s
 R002 If TAd<60 then conclude signes_clin_grav is s
 R003 If FC>140 then conclude signes_clin_grav is s
 R004 If FR>30 then conclude signes_clin_grav is s
 R005 If alt_GI then conclude no(tract_OR) is s
 R006 If trans_degl then conclude no(tract_OR) is s
 R007 If no(alt_GI) and no(trans_degl) then conclude tract_OR is s
 R008 If no(tract_OR) then conclude tract_parenteral is mp
End deductive
Control knowledge
 Deductive control:
```

```
   M001 If K(=(estat_malalt, $x ),s) and member($x,(lleu or mod_g))
             then conclude K(not(shock_septic),s)
   M002 If K(=(TAs,$x),s) and ge($x,90) and K(=(estat_malalt, $y ),s)
             and member($y,(greu or molt_g))
             then conclude K(not(shock_septic),s)
   M004 If K(=(estat_malalt, $x ),s) and member($x,(lleu or mod_g))
             then conclude K(not(alt_neurol),s)
   M005 If K(=(estat_malalt, $x ),s) and member($x,(greu or molt_g))
            then conclude K(not(alt_GI),s)
   M006 If K(=(estat_malalt, $x ),s) and member($x,(greu or molt_g))
            then conclude K(not(trans_degl),s)
End control
End

MODULE ANALITICA=
;; MODUL QUE PREGUNTA DADES ANALITIQUES
  Begin
 Inherit ant
 Inherit situacio_clinica
 Import sodi, hematocrit, leucocits, granulocits, urea, creatinina, pO2
 Export hematocrit, leucocits, granulocits, sodi, urea, creatinina, pO2,
         insuf_resp, insuf_renal_ag, insuf_resp_greu, signes_anal_grav
 Deductive knowledge
 Dictionary:
Predicates:
  sodi= name: "xifra de sodi en sang"
       question: "quin es el valor del sodi en sang?"
       type: numeric
  hematocrit= name:" hematocrit"
             question: "quin es el valor del hematocrit?"
             type: numeric
  leucocits= name:" nombre de leucocits"
            question: "quina es la xifra de leucocits?"
            type: numeric
  granulocits= name: "nombre de segmentats mes bandes"
              question:"quin es el nombre de segmentats mes bandes?"
              type: numeric
              relation: needs leucocits
  creatinina= name: "creatinina"
             question:"quina es la xifra de creatinina en mmol/l?"
             type: numeric
  urea= name: "urea"
       question: "quina es la xifra d'urea en mmol/l?"
       type: numeric
  pO2= name: " presion parcial d'oxigen"
      question: "quina es la pO2 basal en mmHG?"
      type: numeric
  insuf_renal_ag= name: "insuficiencia renal aguda"
                 type: logic
  insuf_resp= name:" insuficiencia respiratoria aguda"
             type: logic
             relation: needs situacio_clinica/estat_malalt
  insuf_resp_greu= name: "insuficiencia respiratoria greu"
                  type: logic
                  relation: needs situacio_clinica/estat_malalt
  signes_anal_grav= name: "signes analitics de gravetat"
                   type: logic
```

```
  granulopenia= name:"granulopenia"
               type:logic
Rules:
  R001 If hematocrit <30 then conclude signes_anal_grav is s
  R002 If granulocits<1000 then conclude granulopenia is s
  R003 If granulopenia then conclude signes_anal_grav is s
  R004 If sodi < 130 then conclude signes_anal_grav is s
  R005 If no(ant/tipus_mal_cron_assoc=(insuf_renal_cron)) and urea> 16.6
           then conclude insuf_renal_ag is s
  R006 If no(ant/tipus_mal_cron_assoc=(insuf_renal_cron)) and creatinina > 220
           then conclude insuf_renal_ag is s
  R007 If insuf_renal_ag then conclude signes_anal_grav is s
  R008 If no(situacio_clinica/estat_malalt=(lleu)) and
            situacio_clinica/FR >24 and pO2 < 60
           then conclude insuf_resp is s
  R009 If no(situacio_clinica/estat_malalt=(lleu or mod_g))
            and situacio_clinica/FR >24 and pO2 < 50
            then conclude insuf_resp_greu is s
 End deductive
  End

MODULE COMP=
;; MODUL QUE PREGUNTA SOBRE COMPLICACIONS
  Begin
 Inherit situacio_clinica
 Import comp_sept, embass, emp, cav, afect_mult, afect_radiol_ext
 Export comp_sept, embass, emp, cav, afect_mult, afect_radiol_ext
 Deductive Knowledge
 Dictionary:
predicates:
  comp_sept= name: "Complicacions septiques"
            question: "s'observan altres focos d'infeccio associats com
                          artritis, meningitis, endocarditis?"
            type: boolean
             relation: needs situacio_clinica/estat_malalt
  emp= name: "empiema"
      question: "l' embassament pleural te criteris d'empiema pleural?"
      type: boolean
      relation: needs embass
  embass= name: "embassament pleural"
         question: " hi ha embassament pleural ?"
         type: boolean
         relation: needs situacio_clinica/estat_malalt
  cav= name: "cavitacio "
      question: "la radiografia de torax mostra cavitacio?"
      type: boolean
      relation: needs situacio_clinica/estat_malalt
  afect_radiol_ext= name: "afectacio simultanea de mes de dos lobuls pulmonars"
                   question: "la RX de torax mostra afectacio de mes de
                             dos lobuls pulmonars?"
                   type: boolean
                   relation: needs afect_mult
                    relation: needs situacio_clinica/estat_malalt
  afect_mult= name:" afectacio multilobar"
             question: "la radiografia de torax mostra afectacio
                       simultanea de dos lobuls pulmonars?"
             type: boolean
```

```
            relation: needs situacio_clinica/estat_malalt
 End deductive
 Control knowledge
  Deductive control:
  M001 If K(not(embass),s) then conclude K(not(emp),s)
  M002 If K(=(situacio_clinica/estat_malalt, (lleu) ),s)
         then conclude K(not(comp_sept),s)
  M003 If K(=(situacio_clinica/estat_malalt, (lleu) ),s)
         then conclude K(not(embass),s)
  M004 If K(=(situacio_clinica/estat_malalt, (lleu) ),s)
        then conclude K(not(cav),s)
  M005 If K(=(situacio_clinica/estat_malalt, (lleu) ),s)
        then conclude K(not(afect_mult),s)
  M006 If K(not(afect_mult),s) then conclude K(not(afect_radiol_ext),s)
  M007 If K(=(situacio_clinica/estat_malalt, $x),s) and
            member ($x, (lleu or mod_g))
           then conclude K(not (afect_radiol_ext),s)
 End control
 End

MODULE CRITERIS_PNEUMONIA_GREU=
;; MODUL PER DEDUIR SI HI HAN CRITERIS DE PNEUMONIA GREU
  Begin
 Inherit situacio_clinica
 Inherit analitica
 Inherit comp
 Export crit_pneum_greu
 Deductive knowledge
 Dictionary:
  Predicates:
   crit_pneum_greu= name: "criteris de pneumonia greu"
                   type: logic
  Rules:
     R001 If situacio_clinica/shock_septic then conclude crit_pneum_greu is s
     R002 If situacio_clinica/alt_neurol then conclude crit_pneum_greu is s
     R003 If analitica/insuf_resp_greu then conclude crit_pneum_greu is s
     R004 If comp/afect_radiol_ext then conclude crit_pneum_greu is s
 End deductive
  End

Module generar_com(x:antimicrobians_general; y:antimicrobians_general)
                                : antimicrobians_general=
 Begin
  Export peni_procaina, peni_G_Na, peni_G_Na_DA, peni_amp_espectre, cloxa,
          ampi, amoxi, eritro_DB, eritro_DA, roxi, imip, amoxi_clav_DB,
          amoxi_clav_DA, ticar_clav, cefuro_OR, cefuro_EV, ceftriax, cefazol,
          cefra, cefmet, cefoxi, ceftaz, clinda_DB, clinda_DA, cipro, oflox,
        tetras_ac_rap, doxi, doxi_DI, cotri_DB, cotri_DI, vanco_tract,
          teico_tract, amika, genta, aztreo, metro_tract, RFM_DA, GCV_tract,
          ACV_DB, ACV_DA, ARA_A_tract, RBV_tract, AMD_DB, AMD_DA, RMD_tract
  Control knowledge
  Evaluation type: eager
  Deductive control:
   ;;creacio de la relacio subsumeix a partir de la relacio belongs_to
   M002 If belongs_to_group($x,$z) and belongs_to_group($y,$z) and diff($x,$y)
          then conclude subsumeix($x,$y)
   ;;Mateixa exportacio de dos submoduls
```

```
 M003 If K(x/$c,int($tc11,$tc12)) and K(y/$c,int($tc21,$tc22))
        then conclude WK($c ,and2(int($tc11,$tc12),int($tc21,$tc22)))
 ;;Cocktails de tipus 11 diferents
 M004 If K(x/$x,int($tc11,$tc12)) and K(y/$y,int($tc21,$tc22)) and atom($x)
           and atom($y) and diff($x,$y) and no(subsumeix($x,$y)) and
        no(subsumeix($y,$x)) and no(espectre_equivalent($x,$y))
        then conclude WK(($x plus $y) ,and2(int($tc11,$tc12),int($tc21,$tc22)))
  ;;potser es perd alguna combinacio amb el subsumir!
;;Cocktails de tipus 21
M005 If K(x/($x plus $y),int($tc11,$tc12)) and K(y/$x,int($tc21,$tc22))
          and atom($x)
       then conclude WK(($x plus $y) ,and2(int($tc11,$tc12),int($tc21,$tc22)))
M006 If K(x/($x plus $y),int($tc11,$tc12))
                        and K(y/$y,int($tc21,$tc22)) and atom($y)
then conclude WK(($x plus $y) ,and2(int($tc11,$tc12),int($tc21,$tc22)))
M007 If K(x/($x plus $y),int($tc11,$tc12)) and K(y/$z,int($tc21,$tc22))
          and atom($z) and diff($z,$x) and diff($z,$y)
          and no(subsumeix($x,$z)) and no(subsumeix($y,$z))
          and no(espectre_equivalent($x,$z))
       and no(espectre_equivalent($y,$z))
       then conclude WK(($x plus ($y plus $z))
                   ,and2(int($tc11,$tc12),int($tc21,$tc22)))
M008 If K(y/($x plus $y),int($tc11,$tc12)) and K(x/$x,int($tc21,$tc22))
                        and atom($x)
then conclude WK(($x plus $y) ,and2(int($tc11,$tc12),int($tc21,$tc22)))
M009 If K(y/($x plus $y),int($tc11,$tc12)) and K(x/$y,int($tc21,$tc22))
                        and atom($y)
then conclude WK(($x plus $y) ,and2(int($tc11,$tc12),int($tc21,$tc22)))
M010 If K(y/($x plus $y),int($tc11,$tc12)) and K(x/$z,int($tc21,$tc22))
          and atom($z) and diff($z,$x) and diff($z,$y) and no(subsumeix($x,$z))
          and no(subsumeix($y,$z)) and no(espectre_equivalent($x,$z))
       and no(espectre_equivalent($y,$z))
       then conclude WK(($x plus ($y plus $z))
                         ,and2(int($tc11,$tc12),int($tc21,$tc22)))
;;Cocktails de tipus 22
 M011 If K(x/($x plus $y),int($tc11,$tc12)) and K(y/($x plus $z),int($tc21,$tc22))
          and atom($z) and atom($y) and diff($z,$y) and no(subsumeix($y,$z))
       and no(subsumeix($z,$y)) and no(espectre_equivalent($y,$z))
       then conclude WK(($x plus ($y plus $z))
                         ,and2(int($tc11,$tc12),int($tc21,$tc22)))
M012 If K(x/($x plus $y),int($tc11,$tc12)) and
         K(y/($z plus $y),int($tc21,$tc22)) and atom($z) and
       atom($x) and diff($z,$x) and no(subsumeix($x,$z)) and
       no(subsumeix($z,$x)) and no(espectre_equivalent($z,$x))
       then conclude WK(($x plus ($y plus $z))
                        ,and2(int($tc11,$tc12),int($tc21,$tc22)))
M013 If K(y/($x plus $y),int($tc11,$tc12)) and K(x/($x plus $z),int($tc21,$tc22))
          and atom($z) and atom($y) and diff($z,$y) and no(subsumeix($y,$z))
       and no(subsumeix($z,$y)) and no(espectre_equivalent($y,$z))
       then conclude WK(($x plus ($y plus $z))
                         ,and2(int($tc11,$tc12),int($tc21,$tc22)))
M014 If K(y/($x plus $y),int($tc11,$tc12)) and K(x/($z plus $y),int($tc21,$tc22))
          and atom($z) and atom($x) and diff($z,$x) and no(subsumeix($x,$z))
          and no(subsumeix($z,$x)) and no(espectre_equivalent($z,$x))
       then conclude WK(($x plus ($y plus $z))
                          ,and2(int($tc11,$tc12),int($tc21,$tc22)))
End control
```

```
  End

Module eliminar_com(x:antimicrobians_general) : antimicrobians_general=
  Begin
  Export peni_procaina, peni_G_Na, peni_G_Na_DA, peni_amp_espectre, cloxa,
          ampi, amoxi, eritro_DB, eritro_DA, roxi, imip, amoxi_clav_DB,
          amoxi_clav_DA, ticar_clav, cefuro_OR, cefuro_EV, ceftriax, cefazol,
          cefra, cefmet, cefoxi, ceftaz, clinda_DB, clinda_DA, cipro, oflox,
        tetras_ac_rap, doxi, doxi_DI, cotri_DB, cotri_DI, vanco_tract,
          teico_tract, amika, genta, aztreo, metro_tract, RFM_DA, GCV_tract,
          ACV_DB, ACV_DA, ARA_A_tract, RBV_tract, AMD_DB, AMD_DA, RMD_tract
  Control knowledge
  Evaluation type: eager
  Deductive control:
   M001 If K(x/$x,int($tc1,$tc2)) then conclude K($x,int($tc1,$tc2))
    M006 If K(($x plus $y),$V) and belongs_to($y,administracio_oral) and
             belongs_to($x,administracio_parenteral)
          then conclude K(($x plus $y) ,int(gp,s))
   M007 If K(($x plus $y),$V) and belongs_to($x,administracio_oral)
             and belongs_to($y,administracio_parenteral)
          then conclude K(($x plus $y) ,int(gp,s))
   ;;IMPROVEMENT No es combinaran antibiotics bacteriostatics
   ;; amb bactericides
   M008 If K(($x plus $y),$V) and belongs_to($y,bacteriostatics)
             and belongs_to($x,bactericides)
             then conclude K(($x plus $y) ,int(gp,s))
   M009 If K(($x plus $y),$V) and belongs_to($x,bacteriostatics)
             and belongs_to($y,bactericides)
          then conclude K(($x plus $y) ,int(gp,s))
   ;;IMPROVEMENT Nomes es matxequen si la certesa de la monoterapia
   ;; esta per sobre de mod-p.
   M010 If K(($x plus $y),int($tc11,$tc12)) and K($y,int($tc21,$tc22))
             AND GT($tc21,modp)
          then conclude K(($x plus $y) ,int(gp,s))
   M011 If K(($x plus $y),int($tc11,$tc12)) and
          K($y,int($tc21,$tc22)) AND LE($tc21,modp) AND LE($tc11, $tc21)
         then conclude K(($x plus $y) ,int(gp,s))
   M012 If K(($x plus $y),int($tc11,$tc12)) and K($x,int($tc21,$tc22))
             AND GT($tc21,modp)
          then conclude K(($x plus $y) ,int(gp,s))
    M013 If K(($x plus $y),int($tc11,$tc12)) and K($x,int($tc21,$tc22))
             AND LE($tc21,modp) AND LE($tc11, $tc21)
          then conclude K(($x plus $y) ,int(gp,s))
  End control
  End

   Module combmycochlam=
       eliminar_com(generar_com(pneumonia_mycoplasma_tractament_2,
                             pneumonia_chlamydia_psit_tractament_2))
```

## C.2 Fuzzy Control Example

Here there is the complete code of the example given in the Section 6.5.

### C.2.1 Controller

We use four modules to implement the controller: *Data*, *Defuzzifier*, *Fuzzy_Inference* and *Fuzzifier*. The module *Data* imports the data of the problem relative to the fuzzifier (slope and width) and the physical measures of the system (the level in the second tank and its variation). The module *Defuzzifier* exports the quantitative value of the control.

```
Module Data =
   Begin
   Import s, w, reference, h2b, deltah2b
   Export s, w, reference, h2b, deltah2b
   Deductive knowledge
      Dictionary:
      Predicates:
      s = Name: "Slope"
          Question: "Slope?"
          Type: Numeric
      w = Name: "Width"
          Question: "Width?"
          Type: Numeric
      reference = Name: "Reference Level"
                  Question: "Reference Level?"
                  Type: Numeric
      h2b = Name: "Level in the second tank"
            Question: "Level in the second tank?"
            Type: Numeric
      deltah2b = Name: "Variation of the Level in the second tank"
                 Question: "Variation of the Level in the second tank?"
                 Type: Numeric
   End Deductive
  End

Module Defuzzifier =
   Begin
   Inherit Data
   Inherit Fuzzy_Inference
   Export v
   Deductive knowledge
      Dictionary:
      Types:
          Q_domain = (PL, PM, PS, PO, NO, NS, NM, NL)
      Predicates:
      v = Name: "Value"
          Type: Numeric
          Function:
          (lambda ()
            (let* ((slope (fact_value Data/s))
                   (terms (type Fuzzy_Inference/Var_u))
                   (values (mapcar
                            (function
                             (lambda (x) (list (first x)
                                               (first (second x)))))
                            (fact_value Fuzzy_Inference/Var_u)))
                   (ling_terms (linguistic_terms))
                   (width (fact_value Data/w)))
              (labels
```

```
((real_value (ling_term)
   (division (position ling_term ling_terms)
             (- (length ling_terms) 1)))
 (reflevel (term)
   (* (-  (+ 1 (position term terms))
          (division (length terms) 2))
      width))
 (recursive_zones (values)
   (let* ((value (first values))
          (rest_values (cdr values))
          (pla_value (center_A_n (second value)
                                 (third value))))
     (cond
      (rest_values
       (let ((next (recursive_zones rest_values))
             (inter_value (center_union_A_n_A_np1
                            (second value)
                            (third value)
                            (third (first rest_values)))))
         (list (+ (first pla_value)
                  (first inter_value)
                  (first next))
               (+ (second pla_value)
                  (second inter_value)
                  (second next)))))
      (t pla_value))))
 (int_xmpot (pot sup inf)
   (division (- (expt sup (+ pot 1)) (expt inf (+ pot 1)))
             (+ pot 1)))
 (int_constant (constant sup inf)
   (* constant (int_xmpot 0 sup inf)))
 (int_xmconstant (constant sup inf)
   (* constant (int_xmpot 1 sup inf)))
 (center_A_n (where A_n)
   (let* ((origin (+ (- where width) slope))
          (final (- where slope)))
     (list (int_xmconstant A_n origin final)
           (int_constant A_n origin final))))
 (center_union_A_n_A_np1 (where A_n A_np1)
   (let* ((X_A_n (+ where (* (- (* 2 A_n) 1) slope)))
          (X_A_np1 (+ where (* (- 1 (* 2 A_np1)) slope)))
          (X_A_n_A_np1 (+ where (* (- 1 (* 2 A_n)) slope)))
          (X_A_np1_A_n (+ where (* (- (* 2 A_np1) 1) slope)))
          (origin (- where slope))
          (final (+ where slope)))
     (labels ((int_down (sup inf)
                (+ (* 0.5 (int_xmpot 0 sup inf))
                   (* (* 0.5 slope)
                      (- (int_xmpot 1 sup inf)
                         (* where (int_xmpot 0 sup inf))))))
              (int_xmdown (sup inf)
                (+ (* 0.5 (int_xmpot 1 sup inf))
                   (* (* 0.5 slope)
                      (- (int_xmpot 2 sup inf)
                         (* where (int_xmpot 1 sup inf))))))
              (int_up (sup inf)
                (- (* 0.5 (int_xmpot 0 sup inf))
```

```
                        (* (* 0.5 slope)
                           (- (int_xmpot 1 sup inf)
                              (* where (int_xmpot 0 sup inf))))))
                   (int_xmup (sup inf)
                     (- (* 0.5 (int_xmpot 1 sup inf))
                        (* (* 0.5 slope)
                           (- (int_xmpot 2 sup inf)
                              (* where (int_xmpot 1 sup inf)))))))
            (cond
             ((and (greateq A_n 0.5)
                   (greateq A_np1 0.5))
              (list (+ (int_xmconstant A_n origin X_A_n)
                       (int_xmdown X_A_n where)
                       (int_xmup where X_A_np1)
                       (int_xmconstant A_np1 X_A_np1 final))
                    (+ (int_constant A_n origin X_A_n)
                       (int_down X_A_n where)
                       (int_up where X_A_np1)
                       (int_constant A_np1 X_A_np1 final))))
             ((and (greateq A_n A_np1)
                   (lesseq A_np1 0.5))
              (list (+ (int_xmconstant A_n origin X_A_n)
                       (int_xmdown X_A_n X_A_np1_A_n)
                       (int_xmconstant A_np1 X_A_np1_A_n final))
                    (+ (int_constant A_n origin X_A_n)
                       (int_down X_A_n X_A_np1_A_n)
                       (int_constant A_np1 X_A_np1_A_n final))))
             ((and (lesseq A_n 0.5)
                   (greateq A_np1 A_n))
              (list (+ (int_xmconstant A_n origin X_A_n_A_np1)
                       (int_xmup X_A_n_A_np1 X_A_np1)
                       (int_xmconstant A_np1 X_A_np1 final))
                    (+ (int_constant A_n origin X_A_n_A_np1)
                       (int_up X_A_n_A_np1 X_A_np1)
                       (int_constant A_np1 X_A_np1 final)))))))))
     (let ((result
            (recursive_zones
             (mapcar
              (function
               (lambda (term)
                 (list term (reflevel term)
                       (let ((exists
                              (position term
                                        (mapcar
                                          (function first)
                                          values))))
                         (cond
                          (exists
                           (real_value (second
                                        (nth exists values))))
                          (t 0))))))
              terms))))
       (division (first result) (second result))))))
End deductive
Control Knowledge
      Evaluation type: eager
End control
```

```
   End

Module Fuzzy_Inference =
   Begin
   Module F = Fuzzifier
   Export Var_u
   Deductive knowledge
   Dictionary:
      Types:
          Q_domain = (PL, PM, PS, P0, N0, NS, NM, NL)
      Predicates:
          Var_u = Name: "Qualitative ACtion"
                  Type: Q_domain
      Rules:
        R001 IF F/e int (NL) and F/Var_e int (NL) THEN conclude Var_u = (PL) is s
        R002 IF F/e int (NL) and F/Var_e int (NM) THEN conclude Var_u = (PL) is s
        R003 IF F/e int (NL) and F/Var_e int (NS) THEN conclude Var_u = (PL) is s
        R004 IF F/e int (NL) and F/Var_e int (N0) THEN conclude Var_u = (PL) is s
        R005 IF F/e int (NL) and F/Var_e int (P0) THEN conclude Var_u = (PL) is s
        R006 IF F/e int (NL) and F/Var_e int (PS) THEN conclude Var_u = (PL) is s
        R007 IF F/e int (NL) and F/Var_e int (PM) THEN conclude Var_u = (P0) is s
        R008 IF F/e int (NL) and F/Var_e int (PL) THEN conclude Var_u = (P0) is s

        R009 IF F/e int (NM) and F/Var_e int (NL) THEN conclude Var_u = (PL) is s
        R010 IF F/e int (NM) and F/Var_e int (NM) THEN conclude Var_u = (PL) is s
        R011 IF F/e int (NM) and F/Var_e int (NS) THEN conclude Var_u = (PM) is s
        R012 IF F/e int (NM) and F/Var_e int (N0) THEN conclude Var_u = (PM) is s
        R013 IF F/e int (NM) and F/Var_e int (P0) THEN conclude Var_u = (PM) is s
        R014 IF F/e int (NM) and F/Var_e int (PS) THEN conclude Var_u = (PS) is s
        R015 IF F/e int (NM) and F/Var_e int (PM) THEN conclude Var_u = (P0) is s
        R016 IF F/e int (NM) and F/Var_e int (PL) THEN conclude Var_u = (P0) is s

        R017 IF F/e int (NS) and F/Var_e int (NL) THEN conclude Var_u = (PL) is s
        R018 IF F/e int (NS) and F/Var_e int (NM) THEN conclude Var_u = (PM) is s
        R019 IF F/e int (NS) and F/Var_e int (NS) THEN conclude Var_u = (PS) is s
        R020 IF F/e int (NS) and F/Var_e int (N0) THEN conclude Var_u = (PS) is s
        R021 IF F/e int (NS) and F/Var_e int (P0) THEN conclude Var_u = (PS) is s
        R022 IF F/e int (NS) and F/Var_e int (PS) THEN conclude Var_u = (P0) is s
        R023 IF F/e int (NS) and F/Var_e int (PM) THEN conclude Var_u = (NS) is s
        R024 IF F/e int (NS) and F/Var_e int (PL) THEN conclude Var_u = (NS) is s

        R025 IF F/e int (N0) and F/Var_e int (NL) THEN conclude Var_u = (PM) is s
        R026 IF F/e int (N0) and F/Var_e int (NM) THEN conclude Var_u = (PM) is s
        R027 IF F/e int (N0) and F/Var_e int (NS) THEN conclude Var_u = (PS) is s
        R028 IF F/e int (N0) and F/Var_e int (N0) THEN conclude Var_u = (P0) is s
        R029 IF F/e int (N0) and F/Var_e int (P0) THEN conclude Var_u = (N0) is s
        R030 IF F/e int (N0) and F/Var_e int (PS) THEN conclude Var_u = (NS) is s
        R031 IF F/e int (N0) and F/Var_e int (PM) THEN conclude Var_u = (NM) is s
        R032 IF F/e int (N0) and F/Var_e int (PL) THEN conclude Var_u = (NM) is s

        R033 IF F/e int (P0) and F/Var_e int (NL) THEN conclude Var_u = (PM) is s
        R034 IF F/e int (P0) and F/Var_e int (NM) THEN conclude Var_u = (PM) is s
        R035 IF F/e int (P0) and F/Var_e int (NS) THEN conclude Var_u = (PS) is s
        R036 IF F/e int (P0) and F/Var_e int (N0) THEN conclude Var_u = (P0) is s
        R037 IF F/e int (P0) and F/Var_e int (P0) THEN conclude Var_u = (N0) is s
        R038 IF F/e int (P0) and F/Var_e int (PS) THEN conclude Var_u = (NS) is s
        R039 IF F/e int (P0) and F/Var_e int (PM) THEN conclude Var_u = (NM) is s
```

```
      R040 IF F/e int (P0) and F/Var_e int (PL) THEN conclude Var_u = (NM) is s

      R041 IF F/e int (PS) and F/Var_e int (NL) THEN conclude Var_u = (PM) is s
      R042 IF F/e int (PS) and F/Var_e int (NM) THEN conclude Var_u = (PS) is s
      R043 IF F/e int (PS) and F/Var_e int (NS) THEN conclude Var_u = (N0) is s
      R044 IF F/e int (PS) and F/Var_e int (N0) THEN conclude Var_u = (NS) is s
      R045 IF F/e int (PS) and F/Var_e int (P0) THEN conclude Var_u = (NS) is s
      R046 IF F/e int (PS) and F/Var_e int (PS) THEN conclude Var_u = (NS) is s
      R047 IF F/e int (PS) and F/Var_e int (PM) THEN conclude Var_u = (NM) is s
      R048 IF F/e int (PS) and F/Var_e int (PL) THEN conclude Var_u = (NL) is s

      R049 IF F/e int (PM) and F/Var_e int (NL) THEN conclude Var_u = (PS) is s
      R050 IF F/e int (PM) and F/Var_e int (NM) THEN conclude Var_u = (N0) is s
      R051 IF F/e int (PM) and F/Var_e int (NS) THEN conclude Var_u = (NS) is s
      R052 IF F/e int (PM) and F/Var_e int (N0) THEN conclude Var_u = (NM) is s
      R053 IF F/e int (PM) and F/Var_e int (P0) THEN conclude Var_u = (NM) is s
      R054 IF F/e int (PM) and F/Var_e int (PS) THEN conclude Var_u = (NM) is s
      R055 IF F/e int (PM) and F/Var_e int (PM) THEN conclude Var_u = (NL) is s
      R056 IF F/e int (PM) and F/Var_e int (PL) THEN conclude Var_u = (NL) is s

      R057 IF F/e int (PL) and F/Var_e int (NL) THEN conclude Var_u = (P0) is s
      R058 IF F/e int (PL) and F/Var_e int (NM) THEN conclude Var_u = (PS) is s
      R059 IF F/e int (PL) and F/Var_e int (NS) THEN conclude Var_u = (NS) is s
      R060 IF F/e int (PL) and F/Var_e int (N0) THEN conclude Var_u = (NL) is s
      R061 IF F/e int (PL) and F/Var_e int (P0) THEN conclude Var_u = (NL) is s
      R062 IF F/e int (PL) and F/Var_e int (PS) THEN conclude Var_u = (NL) is s
      R063 IF F/e int (PL) and F/Var_e int (PM) THEN conclude Var_u = (NL) is s
      R064 IF F/e int (PL) and F/Var_e int (PL) THEN conclude Var_u = (NL) is s
   End deductive
   Control Knowledge
         Evaluation type: eager
   End control
   End

Module Fuzzifier =
   Begin
   Inherit Data
   Export e, Var_e
   Deductive knowledge
      Dictionary:
      Types:
          Q_domain = (PL, PM, PS, P0, N0, NS, NM, NL)
      Predicates:
      e = Name: "Qualitative Value"
          Type: Q_domain
          Function:
          (lambda ()
            (let* ((slope (fact_value Data/s))
                   (terms (type e))
                   (ratio (* (- (division (fact_value Data/h2b)
                                          (fact_value Data/reference))
                                1)
                            250))
                   (ling_terms (linguistic_terms))
                   (width (fact_value Data/w))
                   (Num_terms (length terms)))
              (labels ((fuzzy_value (level)
```

```
                   (nth (truncate
                         (* (length ling_terms)
                            (division  (- ratio (- level slope))
                                       (* 2 slope))))
                        ling_terms))
                 (duplicate (x)
                   (list x x)))
          (dolist (term terms)
            (let ((reflevel
                   (* (-  (+ 1 (position term terms))
                          (division Num_terms 2))
                      width)))
              (cond ((or (eq (car (last terms)) term)
                         (lessthan ratio (- reflevel slope)))
                     (return
                      (list (list term
                                  (car (last ling_terms))))))
                    ((lessthan ratio (+ reflevel slope))
                     (return
                      (list
                       (list
                        term
                        (duplicate
                         (nth (-
                               (- (length ling_terms)
                                  (position (fuzzy_value reflevel)
                                            ling_terms)) 1)
                              ling_terms)))
                       (list
                        (nth  (+ 1 (position term terms)) terms)
                        (duplicate (fuzzy_value reflevel)))))))))))
Var_e = Name: "Qualitative Value"
     Type: Q_domain
     Function:
     (lambda ()
       (let* ((slope (fact_value Data/s))
              (terms (type e))
              (ratio (fact_value Data/deltah2b))
              (ling_terms (linguistic_terms))
              (width (fact_value Data/w))
              (Num_terms (length terms)))
         (labels ((fuzzy_value (level)
                   (nth (truncate
                         (* (length ling_terms)
                            (division  (- ratio (- level slope))
                                       (* 2 slope))))
                        ling_terms))
                 (duplicate (x)
                   (list x x)))
          (dolist (term terms)
            (let ((reflevel
                   (* (-  (+ 1 (position term terms))
                          (division Num_terms 2))
                      width)))
              (cond ((or (eq (car (last terms)) term)
                         (lessthan ratio (- reflevel slope)))
                     (return
```

```
                              (list (list term
                                          (car (last ling_terms))))))
                             ((lessthan ratio (+ reflevel slope))
                              (return
                               (list
                                (list
                                 term
                                 (duplicate
                                  (nth (-
                                        (- (length ling_terms)
                                           (position (fuzzy_value reflevel)
                                                     ling_terms)) 1)
                                      ling_terms)))
                                (list
                                 (nth  (+ 1 (position term terms)) terms)
                                 (duplicate (fuzzy_value reflevel)))))))))))
   End deductive
   Control Knowledge
         Evaluation type: eager
   End control
   End
```

### C.2.2 Simulator

We use the following Lisp function to implement the simulation of the process.

```
(defun simulator (h1a h2a S Q Ts)
  (let* ((p1 (- (/ (+ 3 (sqrt 5)) (* 2 S))))
         (p2 (/ (+ -3 (sqrt 5)) (* 2 S)))
         (c2 (/ (- (* h2a (+ (* S p1) 2)) h1a (* Q S p1))
                (* S (- p1 p2))))
         (c1 (- h2a c2 Q))
         (hb1 (+ (* c1 (exp (* p1 Ts))
                    (+ 2 (* S p1)))
                 (* c2 (exp (* p2 Ts))
                    (+ 2 (* S p2)))
                 (* 2 Q)))
         (hb2 (+ (* c1 (exp (* p1 Ts)))
                 (* c2 (exp (* p2 Ts)))
                 Q))
         (deltah2b (+ (* c1 p1 (exp (* p1 Ts)))
                      (* c2 p2 (exp (* p2 Ts))))))
    (list hb1 hb2 deltah2b)))
```

### C.2.3 Whole Process

The following Lisp functions implements the loop simulator–controller. In the case we use a facility of the shell that consist in using a program with the *external mode*. Using this mode the modules import facts by mean of an external function named *name_of_the_module–import*. The argument of this function is the fact to be imported, and it returns the value of that fact. The facts required from outside the ES are exported by mean of another function named *name_of_the_module–export*. Its argument is a list composed by the fact to be exported and its value.

```
(defvar Simulator_h2b 0)
(defvar Simulator_deltah2b 0)
(defvar defuzzifier_v 0)

(defun data-import (fact)
  (case fact
    (s 2.5)
    (w 5.0)
    (reference 800)
    (h2b Simulator_h2b)
    (deltah2b Simulator_deltah2b)))

(defun defuzzifier-export (fact-value)
  (setq defuzzifier_v (second fact-value)))

(defun two-coupled-tanks-process ()
  (setq Simulator_h2b 0)
  (setq Simulator_deltah2b 0)
  (setq defuzzifier_v 0)
  (let* ((h1a 0)
         (h2a 0)
         (action 0))
    (dotimes (x 1000)
      (let ((defuzzifier_v_old action))
        (Execute 'defuzzifier 'v))
        (setq action (+ defuzzifier_v_old defuzzifier_v))
        (format t "(~S ~S ~S ~S ~S ~S)~%"
                h1a h2a Simulator_deltah2b defuzzifier_v action (abs (- h2a 800)))
        (let ((result (simulator
                       h1a
                       h2a
                       50
                       action
                       20)))
          (setq h1a (first result))
          (setq h2a (second result))
          (setq Simulator_h2b (second result))
          (setq Simulator_deltah2b (third result))
          (ResetKB))))))
```

Notice that the function *Execute* queries the module *defuzzifier* for the value of the fact $v$, and the function *ResetKB* return the ES to its original state.

Example of output:

```
(0 0 0 16.674 16.674 800)
(5.618 0.917 0.075 16.674 33.348 799.082)
(15.478 3.559 0.167 16.674 50.022 796.440)
(28.777 8.012 0.255 16.674 66.696 791.987)
(44.955 14.134 0.333 16.674 83.371 785.865)
(63.579 21.729 0.402 16.674 100.045 778.270)
(84.295 30.603 0.461 16.674 116.719 769.396)
(106.801 40.580 0.512 16.674 133.393 759.419)
(130.845 51.504 0.556 16.674 150.067 748.495)
```

## C.3 Polytrees Example

This is the complete code of the belief propagation in bayesian polytrees given in Section 6.6

```
Module POLYTREE =
  Begin
  Export A, B, C, D, E, F, G
  Deductive knowledge
   Dictionary:
   Types:
         dom_A = (a0 or a1)
         dom_B = (b0 or b1 or b2)
         dom_C = (c0 or c1 or c2)
         dom_D = (d0 or d1)
         dom_E = (e0 or e1)
         dom_F = (f0 or f1 or f2)
         dom_G = (g0 or g1 or g2)
   Predicates:
         A =
              Name: "A"
              Type: array [dom_A]
         A_prior =
              Name: "P(A) Prior probability of A"
              Question: "Enter P(A), prior probability for A"
              Type: array [dom_A]
              Relation: prior A
         A_ptr =
              Name: "Pointer to A"
              Type: logic
              Relation: points_to A
         B =
              Name: "B"
              Type: array [dom_B]
         B_prior =
              Name: "P(B) Prior probability of B"
              Question: "Enter P(B), prior probability for B"
              Type: array [dom_B]
              Relation: prior B
         B_ptr =
              Name: "Pointer to B"
              Type: logic
              Relation: points_to B
         C =
              Name: "C"
              Type: array [dom_C]
         C_prior =
              Name: "P(C), prior probability of C"
              Question: "Enter P(C), prior probability for C"
              Type: array [dom_C]
              Relation: prior C
         C_ptr =
              Name: "Pointer to C"
              Type: logic
              Relation: points_to C
         D =
              Name: "D"
```

```
          Type: array [dom_D]
     D_ptr =
          Name: "Pointer to D"
          Type: logic
          Relation: points_to D
     E =
          Name: "E"
          Type: array [dom_E]
     E_ptr =
          Name: "Pointer to E"
          Type: logic
          Relation: points_to E
     E_evid =
          Name: "Evidence for E"
          Question: "Enter evidence for E"
          Type: array [dom_E]
          Relation: evid E
     F =
          Name: "F"
          Type: array [dom_F]
     F_ptr =
          Name: "Pointer to F"
          Type: logic
          Relation: points_to F
     F_evid =
          Name: "Evidence for F"
          Question: "Enter evidence for F"
          Type: array [dom_F]
          Relation: evid F
     G =
          Name: "G"
          Type: array [dom_G]
     G_ptr =
          Name: "Pointer to G"
          Type: logic
          Relation: points_to G
     G_evid =
          Name: "Evidence for G"
          Question: "Enter evidence for G"
          Type: array [dom_G]
          Relation: evid G
  Rules:
     R01 If A_ptr and B_ptr and C_ptr then conclude D_ptr is
                  ((((0.3 0.7) (0.4 0.6) (0.5 0.5))
                    ((0.75 0.25) (0.82 0.18) (0.35 0.65))
                    ((0.45 0.55) (0.8 0.2) (0.1 0.9)))
                   (((0.3 0.7) (0.99 0.01) (1 0))
                    ((0.37 0.63) (0.85 0.15) (0.21 0.79))
                    ((0.45 0.55) (0.99 0.01) (0.27 0.73))))

;; pi_jkl = p (Di / Aj Bk Cl)
;; ((((p0_000 p1_000)  (p0_001 p1_001)  (p0_002 p1_002))
;;    ((p0_010 p1_010)  (p0_011 p1_011)  (p0_012 p1_012))
;;    ((p0_020 p1_020)  (p0_021 p1_021)  (p0_022 p1_022)))
;;   (((p0_100 p1_000)  (p0_101 p1_001)  (p0_102 p1_002))
;;    ((p0_110 p1_010)  (p0_111 p1_011)  (p0_112 p1_012))
;;    ((p0_120 p1_020)  (p0_121 p1_021)  (p0_122 p1_022))))
```

```
        R02 If D_ptr and B_ptr then conclude E_ptr is ((0.75 0.25) (0.55 0.45))
        R03 If D_ptr then conclude F_ptr is ((0.3 0.2 0.5) (0.1 0.5 0.4))
        R04 If D_ptr then conclude G_ptr is ((0.3 0.6 0.1) (0.5 0.2 0.3))
   End deductive
   Control knowledge
       Evaluation type: reified
       Deductive control:
;; ------------------------------------------------
;;  Translation metarules
       M01 If K(implies ($list_of_premises,$conclusion),$matrix) and
              points_to ($conclusion,$child) and
              set_of_instances ($father,
                 conj ( position ($prem,$list_of_premises,$i),
                points_to ($prem,$father)),
                 $list_of_fathers)
           then conclude
              K(cause ($list_of_fathers,$child),$matrix)

      M02 If points_to ($x_ptr,$x) then conclude node($x)
;; ------------------------------------------------
;; Initializes nodes with evidence
      M03 If  evid($x_evid,$x) and K($x_evid,$v)
           then conclude K(lambda ($x),$v)
;; ------------------------------------------------
;; Initializes root nodes
      M04 If prior($x_prior,$x) and K($x_prior,$v)
           then conclude K(pi($x),$v)
;; ------------------------------------------------
;; Lambda propagation
;; Calculates lambda messages for nodes with several fathers.
      M05 If K(cause ($list_of_fathers,$child),$matrix) and
             K(lambda ($child),$lambda_child) and
             position($father_i,$list_of_fathers,$i) and
             set_of_instances($msg,
               conj(position($father_k,$list_of_fathers,$k),
                   neg(equal($k,$i)),
                   K(pi_msg($father_k,$child),$msg)),
               $pi_msgs_fathers_minus_i)

          then conclude K(lambda_msg($child,$father_i),
              matrix_prod ($lambda_child,
                         transpose (matrix_prod%
                           (cartesian_prod%($pi_msgs_fathers_minus_i),
                           reduce_dim ($matrix, $i)))))
;; Calculates lambda messages for nodes with only one father
      M06 If K(cause($list_of_fathers,$child),$matrix) and
             cardinal($list_of_fathers,1) and
             position ($father,$list_of_fathers,$i) and
             K(lambda($child),$lambda_child)
          then conclude K(lambda_msg($child,$father),
                       matrix_prod($lambda_child,transpose($matrix)))
;; lambda update
      M07 If node($father) and
             set_of_instances($msg,K(lambda_msg($child,$father),
                  $msg),$lambda_msgs_children)
          then conclude K(lambda($father),
```

```
                         inner_product($lambda_msgs_children))
;; --------------------------------------------
;; Pi propagation
;; Calculates pi messages for nodes with several children
      M08 If K(cause($list_of_fathers,$child_j),$matrix) and
             position($father,$list_of_fathers,$i) and
             K(pi($father),$pi_father) and
             set_of_instances($msg,
              conj(K(lambda_msg($child_k,$father),$msg),
                  neg (equal($child_k,$child_j))),
               $lambda_msgs_children_minus_j)
          then conclude K(pi_msg($father,$child_j),
                      norm (inner_product
                       (inner_product ($lambda_msgs_children_minus_j),
                                   $pi_father)))
;; Calculates pi messages for nodes with a unique child
      M09 If K(cause($list_of_fathers,$child),$matrix) and
             position($father,$list_of_fathers,$i) and
             K(pi($father),$pi_father) and
             no(set_of_instances ($child_j,
               conj(K(lambda_msg($child_j,$father),$msg),
              neg(equal($child_j,$child))),
           $other_children))
          then conclude K(pi_msg($father,$child),$pi_father)
;; Pi update
      M10 If K(cause($list_of_fathers,$child),$matrix) and
             set_of_instances ($msg,
                conj(position($father_i,$list_of_fathers,$i),
           K(pi_msg($father_i,$child),$msg)),
                     $pi_msgs_fathers)
          then conclude K(pi($child),
                         norm(matrix_prod
                          (cartesian_prod% ($pi_msgs_fathers),
                        $matrix)))
;; --------------------------------------------
;; Belief update
      M11 If K(lambda($x),$lambda_x) and
             K(pi($x),$pi_x)
          then conclude K($x,norm(inner_product($lambda_x,$pi_x)))
  End control
  End
```

# List of Figures

2.1 Example of module declaration. . . . . . . . . . . . . . . . . . . . 21
2.2 Syntax of interfaces. . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.3 Example of module declaration. . . . . . . . . . . . . . . . . . . . 23
2.4 Syntax of modules. . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.5 Hierarchy example. . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.6 Example of generic module definition and application. . . . . . . 30
2.7 Syntax of generic modules. . . . . . . . . . . . . . . . . . . . . . 31
2.8 Example of module used as parameter of a generic module. . . . 31
2.9 Kernel declaration scheme. . . . . . . . . . . . . . . . . . . . . . 33
2.10 Syntax of refinement, contraction and expansion. . . . . . . . . . 34
2.11 Example of generic module definition and application. . . . . . . 35
2.12 Example of module refinement. . . . . . . . . . . . . . . . . . . . 38
2.13 Example of module refinement. . . . . . . . . . . . . . . . . . . . 39
2.14 Visibility example (hidden modules are written in italic). . . . . 40
2.15 Syntax of submodule declarations. . . . . . . . . . . . . . . . . . 42
2.16 Example of *open* module. . . . . . . . . . . . . . . . . . . . . . . 42
2.17 Syntax of sharing. . . . . . . . . . . . . . . . . . . . . . . . . . . 43

3.1 Example of Local logic declaration. . . . . . . . . . . . . . . . . . 49
3.2 Fuzzy set representing the concept *tall*. . . . . . . . . . . . . . . 52
3.3 Imprecision Ordering on $Int(A_4)$. . . . . . . . . . . . . . . . . . . 55
3.4 Weak Uncertainty Ordering on $Int(A_4)$. . . . . . . . . . . . . . . 56
3.5 Mapping example. . . . . . . . . . . . . . . . . . . . . . . . . . . 64
3.6 Logic declaration. . . . . . . . . . . . . . . . . . . . . . . . . . . 65
3.7 Trapezoidal approximation of a fuzzy interval. . . . . . . . . . . . 66
3.8 Truth table declaration for $T_{A_5}$. . . . . . . . . . . . . . . . . . . . 67
3.9 Renaming declaration example. . . . . . . . . . . . . . . . . . . . 68
3.10 Example of logic declaration. . . . . . . . . . . . . . . . . . . . . 69

4.1 Standard Behavior of an ES. . . . . . . . . . . . . . . . . . . . . 72
4.2 Inference Engine Architecture. . . . . . . . . . . . . . . . . . . . 74
4.3 Example of specialization of a KB . . . . . . . . . . . . . . . . . 79
4.4 Deductive declaration into the modules. . . . . . . . . . . . . . . 94
4.5 Example of dictionary declaration . . . . . . . . . . . . . . . . . 95

4.6 Example of function attribute . . . . . . . . . . . . . . . . . . . . . 98
4.7 Example of characteristic function. . . . . . . . . . . . . . . . . . . . 99
4.8 Syntax of the rules. . . . . . . . . . . . . . . . . . . . . . . . . . . 101
4.9 Syntax of the conditions of rules. . . . . . . . . . . . . . . . . . . . 102
4.10 Syntax of the conclusion of rules. . . . . . . . . . . . . . . . . . . . 104

5.1 Control declaration . . . . . . . . . . . . . . . . . . . . . . . . . . 109
5.2 Example of subsumption. . . . . . . . . . . . . . . . . . . . . . . . 114
5.3 Control cycle. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
5.4 Syntax of the premises of metarules. . . . . . . . . . . . . . . . . . 123
5.5 Syntax of the deductive control. . . . . . . . . . . . . . . . . . . . . 124
5.6 Syntax of the structural control. . . . . . . . . . . . . . . . . . . . 125

6.1 Architecture of *Terap–IA* application. . . . . . . . . . . . . . . . . 129
6.2 Example of filtering. . . . . . . . . . . . . . . . . . . . . . . . . . . 132
6.3 Module Renal Failure. . . . . . . . . . . . . . . . . . . . . . . . . . 133
6.4 Module Pneumonia Mycoplasma Treatment. . . . . . . . . . . . . . 135
6.5 Example of module tree. . . . . . . . . . . . . . . . . . . . . . . . . 137
6.6 Case example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138
6.7 Coupled tanks example. . . . . . . . . . . . . . . . . . . . . . . . . 139
6.8 Scheme of the process. . . . . . . . . . . . . . . . . . . . . . . . . . 140
6.9 Fuzzy control modules. . . . . . . . . . . . . . . . . . . . . . . . . . 142
6.10 Fuzzification process. . . . . . . . . . . . . . . . . . . . . . . . . . 142
6.11 Defuzzication by mean of the gravity center. . . . . . . . . . . . . . 145
6.12 Results for $h_2$, $dQ$ and $Q$. . . . . . . . . . . . . . . . . . . . . . . . 146
6.13 Detailed $dQ$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147
6.14 Phase plane result. . . . . . . . . . . . . . . . . . . . . . . . . . . 148
6.15 Node example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148
6.16 Comparations among applications. . . . . . . . . . . . . . . . . . . 156

# List of Tables


1.1 Main differences between *Milord* and **Milord II**. . . . . . . . . . 5

3.1 $T_{S_5}$ Table. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.2 $N_5$ Table. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
3.3 $I_{T_{S_5}}(x, y)$ Table. . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
3.4 $MP_{T_{S_5}}(x, y)$ Table. . . . . . . . . . . . . . . . . . . . . . . . . . . 51
3.5 $T_{Gram_7}$ Table. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

4.1 Main differences between *Milord* and **Milord II** inference engines 80
4.2 Valid models of the example. . . . . . . . . . . . . . . . . . . . . . 85
4.3 Operations between expressions. . . . . . . . . . . . . . . . . . . . 104

6.1 Mac Vicar–Wheland's initial set of rules. . . . . . . . . . . . . . . . 144

# References

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

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

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

Alsina, C., Grané, J., Sales, T. and Trillas, E. (1984). Algunes consideracions sobre el modus ponens: Funcions de modus ponens. In *Actes del III Congrés Català de Lògica Matemàtica. Barcelona*, pages 55–77.

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

Barroso, C. (1992). ENS–AI: un sistema experto para la enseñanza. In *Proceedings of the European Conference about Information Technology in Education: a critical insight. Barcelona.*, volume 2, pages 373–382.

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

Berenji, H. R. (1992). *An Introduction to Fuzzy Logic Applications in Intelligent Systems*, chapter Fuzzy Logic Controllers, pages 69–96. Kluwer Academic Publishers.

Bonissone, P., Gans, S. and Decker, K. (1987). Rum: A layered architecture for reasoning with uncertaintyg. In *IJCAI'87*, pages 891–898.

Chandrasekaran, B. (1986). Generic tasks in knowledge–based reasoning: High–level building blocks for expert systems design. Technical report, Ohio State University.

Chandrasekaran, B. (1987). Towards a functional architecture for intelligence based on generic information processing tasks. In *Proceedings of the IJCAI'87*, pages 1183–1192.

Davis, R. (1982). *Knowledge–Based Systems in Artificial Intelligence*, chapter TEIRESIAS: Applications of Meta–Level Knowledge, pages 920–927. McGraw–Hill, New York.

Dechter, R., Meiri, I. and Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence*, 49:61–95.

Demolombe, R. (1990). Strategies for the computation of conditional answers. In *Proceedings of the Workshop on Partial Deduction, Partial Evaluation and Intelligent Reasoning, ECAI'90*, pages 5–23.

Dempster, A. P. (1967). Upper and lower probabilities induced by a multivalued mapping. *Annals of Mathematical Statistics*, 38:325–339.

Domingo, M. (1993a). Evaluating the expert system approach to biological identification through application to porifera. In *Sponges in Time and Space. Proceedings 4th International Porifera Congress*, page In press.

Domingo, M. (1993b). Towards a knowledge level analysis of classification in biological domains. In *Proceedings of the IMACS International Workshop on Qualitative Reasoning and Decision Technologies QUARDET'93.*, pages 535–544.

Domingo, M. (1995). *An expert system architecture for taxonomic domains. An application in Porifera: the development of Spongia*. PhD thesis, Universitat de Barcelona.

Dubois, D. and Prade, H. (1988). *Possibility Theory: An Approach to Computerized Processing of Uncertainty*. Plenum Press.

Duda, R. O., Hart, P. E. and Nilsson, N. J. (1976). Subjective bayesian methods for rule–based inference systems. In *Proceedings of the AFIPS National Computer Conference*, volume 7, pages 1075–1082.

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

Fensen, D., Angele, J. and Landes, D. (1991). Karl: A knowledge adquisition and representation language. In *Proceedings of the 11th Conference on Expert Systems and their Applications (Avignon)*, pages 513–525.

Forgy, C. (1981). OPS5 user manual. Technical Report CMU–CS–81–135, Computer Science Department, Carnegie Melon University.

Foulloy, L. (1993). Qualitative control and fuzzy control: Towards a writing methodology. *AICOM*, 6(3/4).

Fox, J. (1989). *Knowledge Engeneering*, chapter Symbolic Decision Procedures for Knowledge Based Systems. McGraw Hill.

Gallagher, J. (1986). Transforming logic programming by specialising interpreters. In *Proceedings ECAI'86*, pages 109–122.

Giunchiglia, E., Traverso, P. and Giunchiglia, F. (1993). *Formal Specification of Complex Reasoning Systems*, chapter Multi–Context Systems as a Specification Framework for Complex Reasoning Systems. Ellis Horwood.

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

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

Godo, L. and Meseguer, P. (1991). A constraint–based approach to generate finite truth–values algebras. Technical Report 91/9, IIIa–CEAB.

Goguen, J. A. (1986). Reusing and interconnecting software components. *IEEE Computer*, February:16–28.

Gréboval, C. and Kassel, G. (1992). Modelling at the knowledge level: The shell AIDE. In *Proceedings of the 12th International Conference on Artificial Intelligence, Expert Systems and Natural Language, Avignon, France*.

Hàjek, P., Havrànek, T. and Jirousek, R. (1992). *Processing Uncertain Information in Expert Systems*. CRC Press.

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

Harper, R., Sannella, D. and Tarlecki, A. (1989). Structure and representation in LCF. In *Proceedings of 4th IEEE Symp. on Logic of Computer Science*.

Jonckers, V., Geldof, S. and De Vroede, K. (1992). The COMMET methodology and workbench in practice. Technical Report 92–8, Vrije Universiteit Brussel. Laboratory for Artificial Intelligence.

Kleene, S. (1952). *Introduction to Metamathematics*. Van Nostrand.

Komorowski, H. J. (1981). *A specification of an abstract Prolog machine and its application to partial evaluation*. PhD thesis, Linkoping University.

Komorowski, H. J. (1990). Towards a programming methodology founded on partial deduction. In *Proceedings ECAI'90*, pages 404–409.

Kuipers, B., Moskowitz, A. and Kassirer, J. (1988). Critical decisions under uncertainty: Representation and structure. *Cognitive Science*, 12:177–210.

Langevelde, I. v., Philipsen, A. and Treur, J. (1993). *Formal Specification of Complex Reasoning Systems*, chapter A Compositional Architecture for Simple Design Formally Specified in DESIRE. Ellis Horwood.

Lloyd, J. W. and Shepherson, J. C. (1991). Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3/4):217–242.

López de Mántaras, R. (1990). *Approximate Reasoning Models*. Ellis Horwood Series in Artificial Intelligence.

López, B. (1993). *Aprenentatge i generació de plans per a Sistemes Experts*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona.

Meseguer, P. (1992). *Validation of Multi–Level Rule–Based Expert Systems*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona.

Miller, D. A. (1986). A theory of modules for logic programming. In *AAVV: Proceedings of 1986 IEEE Sympos. on Logic Programming*.

Nilsson, N. J. (1986). Probabilistic logic. *Artificial Intelligence Journal*, 28:71–88.

O'Keefe, R. (1985). Towards an algebra for constructing logic programs. In *AAVV: Proceedings of 1985 IEEE Sympos. on Logic Programming*, pages 152–160.

Pearl, J. (1986). A constrain–propagation approach to probabilistic reasoning. In Kanal, N. L. and Lemmer, J. F., editors, *Uncertainty in Artificial Intelligence*, pages 357–369. North Holland.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Pearl, J. (1990). Reasoning under uncertainty. *Annual Review in Computer Science*, 4:37–42.

Plaza, E. and López de Mántaras, R. (1989). Model–based knowledge acquisition for heuristic classification systems. *SIGART Newsletter*, 108:98–105.

Puyol, J. (1989a). Hacia un modelo de computación concurrente para sistemas expertos. In *Proceedings III Reunión Técnica de la Asociación Española para la Inteligéncia Artificial*, pages 23–31.

Puyol, J. (1989b). Parallel programming in expert systems. In *Proceedings Third World Conference on Mathematics at the Service of Man*.

Puyol, J. (1990). ADES: un entorn per a sistemes experts distribuïts. Master's thesis, Universitat Autònoma de Barcelona.

Puyol, J., Sierra, C. and Agustí, J. (1991). Partial evaluation in MILORD II: A language for knowledge engineering. In *Proceedings Europ–IA'91*, pages 193–207.

Puyol, J. (1992a). An inference engine based on specialisation with uncertainty. In *Proceedings IPMU'92*, pages 725–728.

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

Puyol, J., Godo, L. and Sierra, C. (1992c). A specialisation calculus to improve expert system communication (long paper). Technical Report 92/8, IIIA–CSIC.

Sakama, C. and Itoh, H. (1986). Partial evaluation of queries in deductive databases. Technical Report TR–302, ICOT.

Sannella, D. and Wallen, L. A. (1987). A calculus for the construction of modular prolog programs. In *AAVV: Proceedings of 1987 IEEE Sympos. on Logic Programming*, pages 368–378.

Shafer, G. (1976). *A mathematical theory of the evidence*. Princeton University Press.

Shortliffe, E. H. and Buchanan, B. G. (1975). A model of inexact reasoning in medecine. *Mathematical Biosciencies*, 23:351–379.

Shortliffe, E. H. (1976). *Computer Based Medical Consultations: MYCIN*. American Elsevier, New York.

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

Sierra, C. and Agustí, J. (1991). Colapses: Towards a methodology and a language for knowledge engineering. In *Proceedings AVIGNON'91*, pages 407–423.

Steele, G. (1984). *Common Lisp: The Language*. Digital Press.

Steels, L. (1990). Components of expertise. *AI Magazine*, 11.

Sticklen, J., Smith, J. W., Chandrasekaran, B. and Josephson, J. R. (1987). Modularity of domain knowledge. *International Journal of Expert Systems*, 1(1):1–15.

Takeuchi, A. and Furukawa, K. (1986). Partial evaluation of prolog programs and its application to meta programming. In *Information Processing 86*.

Treur, J. and Wetter, T., editors (1993). *Formal Specification of Complex Reasoning Systems*. Ellis Horwood.

Trillas, E. and Valverde, L. (1987). On inference in fuzzy logic. In *Second IFSA Congress. Tokyo*, pages 294–297.

Turner, R. (1984). *Logics for Artificial Intelligence*. Ellis Horwood Series in Artificial Intelligence.

Valverde, L. and Trillas, E. (1985). On modus ponens in fuzzy logic. In *Proceedings 15th ISMVL. Kignston (Ontario)*, pages 294–301.

van Harmelen, F. and Balder, J. (1992). $(ML)^2$: A formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1).

van Harmelen, F., López de Mántaras, R. and Malec, J. (1993). *Formal Specification of Complex Reasoning Systems*, chapter Comparing Formal Specification Languages for Complex Reasoning Systems, pages 257–282. Ellis Horwood.

Vasey, P. (1986). Qualified answers and their application to transformation. In Goos, G. and Hartmanis, J., editors, *Third International Conference in Logic Programming, LNCS 225*, pages 425–432. Springer–Verlag.

Veld, L., Jonker, W. and Spee, J. (1993). *Formal Specification of Complex Reasoning Systems*, chapter Specifications of Complex Reasoning Tasks in $K_{BS}SF$. Ellis Horwood.

Venken, R. (1984). A prolog meta–interpreter for partial evaluation and its application to source transformation and query–optimisation. In *Proceedings ECAI'84*, pages 91–100.

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

Vicar-Whelan, P. J. M. (1976). Fuzzy sets for man–machine interaction. *International Journal of Man–Machine Studies*, 84:687–697.

Vila, L. (1993a). Constraints on distances between temporal distances. Report de Recerca forthcoming, IIIA.

Vila, L. (1993b). Instants, periods and the divided instant problem. In *proc. of QUARDET'93*. IMACS.

Vila, L. (1993c). A theory of time based on instants and periods. Report de Recerca forthcoming, IIIA. Submitted to 1rst Intl. Conf. on Temporal Logic.

Vila, L. (1993d). Time ontology and temporal occurrence predicates. In *TARRAT'93*. IIIA.

Vila, L. (1995). *On Temporal Representation and Reasoning in Knowledge–Based Systems*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona.

Wielinga, B. J., Schreiber, A. T. and Breuker, J. A. (1992). Kads: A modelling approach to knowledge engineering (special issue). *Knowledge Acquisition*, 4(1).

Wolstenholme, D. (1987). Saying i don't know and conditional answers. In Moralle, D., editor, *Research and Development in Expert Systems IV*, pages 115–125. Cambridge University Press.

Zadeh, L. A. (1965). Fuzzy sets. *Inf. Control*, 8:338–353.

Zadeh, L. A. (1975). Fuzzy logic and approximate reasoning. *Synthese*, 30:407–428.

# Index

*Ens–AI*, 136
*Milord*, 3
*Spong–IA*, 136
*Terap–IA*, 128, 177

Algebra of truth–values, 47

Completeness, 87, 171
Contraction, 32, 40

Deductive Control, 124
Deductive Knowledge, 94
Deductive Process, 74
Dynamic Modules, 43

Eager, 119, 125
Evaluation Strategy, 117
Expansion, 32, 40

Fact Declarations, 95
Fact Types, 96
Fuzzy Control Example, 139, 195
Fuzzy Sets, 57, 98

Generic Modules, 19, 28

Hierarchy of modules, 24

Imprecision, 45, 53
Inference Engine, 88
Information Hiding, 37
Inherit Declaration, 41
Inheritance, 36
Interfaces, 22
Intervals of Truth–values, 54

Lazy, 117, 125
Local Logic Declaration, 65
Local Logics, 60

Metarules, 124
Modules, 18, 25

Open Declaration, 41

Propagation Rules Example, 146, 204

Refinement, 32, 34
Reflection, 119
Reification, 119
Reified, 125
Rule Declarations, 100

Search Process, 73
Search Strategy, 117
Semantics of Specialization, 82
Sharing Declaration, 43
Soundness, 87, 171
Specialization, 9, 76, 90
Specialization Calculus, 81
Structural Control, 124
Subsumption, 109
Syntax of **Milord II**, 161
Syntax of Specialization, 81

Threshold, 116

Uncertainty, 45, 51
Unnecessary Rules, 115

Validation, 79