MONOGRAFIES DE L'INSTITUT D'INVESTIGACIÓ
EN INTEL·LIGÈNCIA ARTIFICIAL
Number 5

Institut d'Investigació
en Intel·ligència Artificial

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

A Framework for Integrated Learning and Problem Solving

Eva Armengol i Voltas

Foreword by Enric Plaza

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
Enric Plaza
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
Eva Armengol i Voltas
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Institut d'Investigació
en Intel·ligència Artificial

ISBN: 84-00-07741-5
Dip. Legal: B-42482-98
© 1998 by Eva Armengol i Voltas

Printed by CPDA-ETSEIB.
Avinguda Diagonal, 647.
08028 Barcelona, Spain.

A en Lluís.

Contents

Foreword

The development of complex knowledge systems has revolved around the hairy issue of knowledge acquisition. This book proposes a leading approach to integrate knowledge modeling an Machine Learning (ML), two of the main research disciplines dealing with the knowledge acquisition issue, in the process of designing and implementing knowledge systems, and specially knowledge systems with learning capabilities.

A main contribution of "A framework for integrated learning and problem solving" is that of using knowledge modeling to analyze ML techniques as "methods". This approach allows analyzing at the knowledge level the learning processes we might be interested in, and then integrate them into the results of the analysis performed by knowledge modeling techniques. In this way, the methodological aspects of knowledge modeling are incorporated to the integration of ML techniques and to the development of learning systems.

Finally, the new learning methods presented in the book are of particular interest for the ML community. The ML methods work upon a representation formalism called feature terms. Feature terms formalize the object-centered representation of NOOS, the language used to develop and implement the framework hereby presented. These ML methods, based on unification of feature terms, show how relational learning can be achieved using an object-centered representation. This contribution opens an exciting new space of possibilities for object-centered induction and lazy learning.

Bellaterra, July 1998

Enric Plaza i Cervera,
IIIA-CSIC
E-mail: enric@iiia.csic.es

Preface

Fer una tesi no és un treball fàcil, però si la gent del teu entorn és com la que jo he trobat a l'IIIA, la feina es fa menys feixuga. A tothom sense excepció li he d'agrair el suport que m'han donat. En particular a Enric Plaza, director d'aquest treball, que és la persona que ha fet possible aquesta recerca gràcies a les idees que m'ha ha aportat i a les llargues discussions que hem mantingut. També vull agrair a Josep Lluís Arcos la seva paciència i la seva companyonia, ajudant-me sempre que ho he necessitat. A Francisco Martin i a Adriana Zapico els hi agraeixo la seva disposició a ajudar-me en qualsevol moment. A Marta Domingo li he d'agrair el seu paper d'experta en esponges, que m'ha servit per desenvolupar una de les aplicacions.

A Lluís Bonamusa li he d'agrair moltes coses: la paciència, el suport, la seva ajuda com expert del domini, com a dibuixant, etc. Conviure amb una persona que està fent la tesi no és una tasca fàcil i ell l'ha aprovat amb escreix.

A Conrad Armengol, el meu pare, li vull agrair el seu suport i la il.lusió que té per tot el que faig. Finalment, vull tenir un record per la meva mare M. del Pilar Voltas, ella no ha pogut veure acabat aquest treball però sempre em va donar el suport necessari.

Aquest treball ha estat finançat pels projectes de recerca AMP (CICYT 801/90C02), ANALOG (CICYT-122/93), SMASH (TIC-1038-C04-01), per la xarxa europea d'excel.lència ML-NET (ESPRIT 7115) i per una beca per a la recerca del Ministeri d'Educació i Cultura.

Bellaterra, Juliol 1998

Eva Armengol i Voltas
IIIA-CSIC
E-mail: eva@iiia.csic.es

Abstract

The proposal of this thesis is the integration of Knowledge Modelling and Machine Learning. Commonly, Knowledge Modelling methodologies have been used to build Knowledge systems (without Machine Learning capabilities). We propose to use Knowledge Modelling methodologies to analyse Machine Learning techniques and their integration into problem solving systems. The result of this proposal is a common framework and implementation for integrated learning and problem solving.

We achieve this proposal by defining a framework modifying some assumptions of the Knowledge Modelling methodologies. In this framework an application domain can be modelled as a task/method decomposition: a problem solving method decomposes a task into subtasks and each subtask has associated problem solving methods that solve it. We propose to use Knowledge Modelling methodologies to analyse Machine Learning techniques. As a result we consider that Machine Learning techniques are learning methods and, consequently, they can be modelled as a task/method decomposition. Learning methods are associated to a particular kind of tasks, called KA-Tasks, whose goal is the acquisition of domain knowledge. Thus, our framework integrates problem solving and learning methods thanks to their uniform representation as a task/method decomposition.

Elements of our framework can be implemented using NOOS, a representation language whose main representation formalism are the *feature terms*. Feature terms provide a uniform representation of both KM analysis and learning methods. Also, feature terms allow the representation of objects belonging to both relational domains (those used in ILP) and propositional domains. Feature terms form a partial order by means of the subsumption relationship. From the subsumption we introduce the anti-unification operation.

Based on the anti-unification concept we introduce three learning methods (INDIE, DISC and LID) handling feature terms. INDIE is a heuristic bottom-up inductive learning method that uses the anti-unification concept and the López de Mántaras distance to generalise descriptions. DISC is a heuristic top-down inductive learning method that uses the anti-unification concept as bias to specialise descriptions. LID is a lazy learning method that uses the Shannon's entropy estimation to retrieve experiences represented in a structured way.

The framework and the methods we propose have been used for developing two applications: CHROMA and SPIN. CHROMA is an application supporting the search for the appropriate plan to purify a protein. SPIN is an application for classifying marine sponges that integrates learning and problem solving.

Chapter 1

Introduction

1. Motivation

Knowledge systems (KS) address problem solving in a specific domain task by the intensive use of domain specific knowledge. The necessary knowledge can be acquired in two ways:

1) From a domain expert during a Knowledge Acquisition phase previous to building the problem solver, or

2) Using Machine Learning techniques such as learning from examples.

During the Knowledge Acquisition phase a domain expert and a knowledge engineer work together to determine the knowledge necessary to solve problems of that domain. The agreement on the vocabulary used by both experts is not an easy task. Several methodologies have been developed to support this agreement. Currently, the more used are the Knowledge Modelling (KM) methodologies, such as KADS (Wielinga et al., 1992), Generic Tasks (Chandrasekaran, 1986), and Components of Expertise (Steels, 1990). These methodologies determine which are the problems to solve (tasks), how can be solved (methods), and which knowledge is necessary (models). Thus, an application domain is analysed in terms of these three basic elements, being its main goal to obtain models for solving problems in a domain.

Knowledge Modelling frameworks analyse the knowledge necessary to solve problems and how it can be used. The use of KM methodologies allows the analysis of a domain in a implementation-independent way, therefore a complementary effort is necessary to implement the acquired knowledge. The main assumption of the KM methodologies is that all the knowledge can be obtained before solving any problem. That is to say, during the Knowledge System (KS) design all the necessary knowledge is determined and acquired, so the resulting KS is ready to solve new problems.

Machine Learning techniques also allow the acquisition of knowledge, nevertheless their assumption is different. These techniques consider two kinds of knowledge: background knowledge and learned knowledge. The background knowledge is usually acquired from the domain expert without using a specific methodology. The learned knowledge is acquired by Machine Learning methods that use the background knowledge and examples (solved problems) to obtain new domain knowledge.

Any domain has some knowledge that can be easily acquired from the domain expert and some other knowledge whose acquisition can be automatically made using Machine Learning techniques. The integration of Machine Learning and Knowledge Modelling is desirable since it allows knowledge acquisition from experts during the modelling phase and also provides an opportunity to learn new knowledge from examples during problem solving.

Nevertheless, the integration of Knowledge Modelling and Machine Learning has to solve issues such as how the KM methodologies have to be modified in order to acquire knowledge during the problem solving or how ML techniques can use KM methodologies to acquire the background knowledge. There is also a question making difficult a practical integration of Knowledge Modelling and Machine Learning: the representation language used. The KM analysis is represented using high level specification languages whereas ML methods use implementation level formalisms.

The proposal of this thesis is to define a framework capable to integrate Knowledge Modelling and Machine Learning. In this framework, problem solving methods (PSM) decompose tasks in subtasks and subtasks have, in turn, associate problem solving methods that solve them. The integration of Knowledge Modelling and Machine Learning is achieved using KM to analyse learning methods. So, learning methods decompose tasks in subtasks and each subtask can be solved using some problem solving method.

Therefore, in the framework we propose, an application task is analysed in terms of a task/method decomposition. This decomposition includes a particular kind of tasks that we call *KA-tasks*, whose goal is the acquisition of knowledge necessary for some problem solving activity. A KA-task is solved using a learning method. In fact, the result of a KA-task (as any task) is the construction of a model that is necessary to apply some problem solving method.

The use of KM methodologies to analyse learning methods allows a seamless integration of problem solving and (symbolic) Machine Learning. Thus, in the framework we propose, an application domain is analysed identifying tasks, models and methods. In particular, two kinds of models can be identified: 1) models that may be acquired from a domain expert and, 2) models that may be acquired using a learning method. The first kind of models plays a role similar to the models in KM methodologies whereas the second kind of models are associated to KA-Tasks. During the KM analysis of a specific domain this second kind of models are identified and, for each one, a KA-Task is defined. For each KA-Task one has to determine the kind of input and output models, which learning method can solve it and when this task can be used. In turn, this KA-Task will be solved during the problem solving using a learning method.

Related issues are how and when the appropriate method to solve a task is selected. Some Knowledge Modelling methodologies such as KADS or Generic Tasks associate one method to solve each task. Nevertheless a task could be used in several moments using different knowledge and, therefore, solved using different methods. For this reason, some other methodologies such as CommonKADS (Wielinga et al., 1993) or Components of Expertise allow the association of several methods to a task. However, the selection of the appropriate method to solve a task is made during the KS design.

Using our framework, several methods can be associated to a task (or KA-Task) during the KM analysis of the domain. The selection of the appropriate method, differently than in CommonKADS or Components of Expertise, can be delayed to the implemented system. We propose that this selection can be made according to the problem to be solved. We say that this kind of selection is *lazy problem-centred*. Thus, the knowledge necessary to decide which is the appropriate method has to be acquired during the design phase. Notice that this framework also supports the design of Multistrategy Learning Systems since during the KM analysis of a domain, more than one KA-Task can be established. Each KA-Task can have a specific, and may be different, learning method.

Elements of the framework we present can be implemented using the reflexive object-centred representation language NOOS (Arcos, 1997). The NOOS language is based on task/method decomposition and provides a uniform representation of domain knowledge, problem solving methods and learning methods. Therefore, the use of NOOS solves the question of the different representation languages used by KM and ML. NOOS structures are near to the structures obtained from the KM analysis of a domain and they also allow the implementation of learning methods.

The representation formalism in which NOOS is based are *feature terms*. Feature terms are a generalisation of first-order terms allowing the representation of object-oriented capabilities into declarative languages. Feature terms can be partially ordered using the *subsumption* relationship. Subsumption is equivalent to the *more general than* relation commonly used in Machine Learning. In fact, if a feature term X subsumes another feature term Y, this means that X is more general than Y. Using the subsumption, feature terms form a lattice.

Based on subsumption we have defined the *anti-unification* operation.

Intuitively, the anti-unification of two feature terms X and Y is a feature term Z containing all that is common to X and Y. The feature term Z is the least general generalisation of X and Y.

The introduction of feature terms has required the definition of new learning methods to deal with them. In ILP, the θ-subsumption is the basis for two strategies, bottom-up and top-down, exploring the hypothesis space. In the same way, based on both subsumption and anti-unification, we have defined two inductive learning methods (INDIE and DISC) and one lazy learning method (LID).

Given a set of training examples represented as feature terms, the goal of INDIE and DISC is to build a feature term representing a concept description. In both methods, induction is viewed as a search (bottom-up in INDIE and top-down in DISC) in the space of feature terms.

The key issue of CBR is the retrieval of past experiences (cases) to solve new problems. Cases are organised according to some indexes in order to retrieve the most useful for each new problem. Commonly, cases are represented as sets of attribute-value and indexes useful for retrieval are based on those attributes. The retrieval consists of applying some metrics to indexes. Using feature terms cases have a structured representation, therefore we need to define some retrieval mechanism taking this into account. LID is a lazy learning method that handles cases represented as feature terms. In this method the retrieval of precedents is based on both the anti-unification and the Shannon's entropy.

The framework and the learning methods above have been used for developing two applications: CHROMA and SPIN. CHROMA is an application supporting the search for the appropriate plan to purify a protein. CHROMA integrates problem solving and learning in a lazy problem-centred way. SPIN is an application for classifying marine sponges that integrates learning and problem solving. INDIE, DISC and LID have been used in SPIN.

2. Structure of the Thesis

This thesis is divided in three parts. In Part I we examine the current state of the art and describe the proposed framework for integrated learning and problem solving. In Part II we present some new learning methods useful for the formalism of feature terms. Finally in Part III we describe some prototype of domain applications developed using the framework and the learning methods respectively described in Parts I and II.

The first part is composed of two chapters:

- Chapter 2 is a state-of-the-art of Knowledge Modelling, Machine learning and Problem Solving. In this chapter all the concepts necessary to describe our framework are introduced. Also we explain the context of our work.

- In chapter 3 we propose a framework for integrating problem solving and learning. This framework has been implemented using NOOS, an object-centred representation language that uses the feature terms formalism to represent objects. NOOS is concisely described in this chapter.

In Part II there are three chapters each one describing a ML method that works using the feature terms formalism.

- In chapter 4 we introduce some concepts on learning in the feature terms formalism, such as anti-unification, common to the methods described in the next chapters.
- In chapter 5 the INDIE method is described and evaluated. INDIE is a heuristic bottom-up inductive learning method that uses the anti-unification concept and a heuristic based on López de Mántaras distance to generalise descriptions.
- In chapter 6 we describe a heuristic top-down inductive learning method called DISC. DISC uses the anti-unification concept as bias to specialise descriptions.
- In chapter 7 a lazy learning method, called LID, is described. LID builds, in a lazy problem-centred way, a discriminant description for a specific problem. The basis of LID are the anti-unification operation and an entropy-reduction heuristic.

In Part III we describe two prototypes developed using the framework described in Part I.

- In chapter 8 we describe CHROMA, an application that recommends a plan purifying a protein. This application shows both the dynamic integration of learning and problem solving, and the lazy-problem centred approach for problem solving.
- In chapter 9 we describe SPIN, an application for classifying marine sponges. This application can use any of the methods presented in Part II to classify a new sponge in the biological taxonomy.

Finally, in chapter 10 we summarise the final conclusions including the contributions of our work as well as the future work.

PART I

Chapter 2

Knowledge Modelling, Learning and Problem Solving

1. Introduction

There are two families of techniques to acquire and organise the knowledge of a Knowledge-based System (KBS): Knowledge Acquisition and Machine Learning. Goals of Knowledge Acquisition (KA) are improving and automating the knowledge acquisition from human experts (knowledge engineering). Conversely, Machine Learning (ML) is focused in the development of algorithms allowing to acquire knowledge from data and also the automatic improvement of the knowledge organisation. The construction of a KBS is composed of three phases (Bareiss et al., 1989): systematic elicitation of the knowledge from the expert, KB refinement and KB reformulation. Each one of these phases is achieved in a different way according to the used (KA or ML) technique. During the knowledge elicitation phase using a Knowledge Acquisition technique, the expert provides the basic terminology and the conceptual structure of a domain. Instead, Machine Learning techniques assume that there is a knowledge representation and a background knowledge before learning.

The advance in the research has found a complementarity between both topics, in the sense that the integration of Machine Learning and

Knowledge Acquisition techniques can make easy the construction of a KBS. There have been several attempts in this direction: interactive knowledge-based assistants that obtain new knowledge from the observation and the analysis of the problem solver, Case-based Reasoning Systems that integrate knowledge elicitation and case refinement with inductive generalisation, or Interactive Inductive Logic Programming Systems that integrate knowledge elicitation from the expert. Advances in the integration of Knowledge Acquisition and Machine learning can be found in (Marcus, 1989).

Research in Machine Learning has analysed several learning methods, such as empirical induction, explanation-based learning (De Jong and Mooney, 1986; Mitchell et al., 1986) or analogy (Carbonell, 1986). Nevertheless none of these methods can deal alone with complex real-world problems. Therefore, current research has been focused in Multistrategic Learning Systems (Michalski and Tecuci, 1991). These systems have available several learning methods that can be applied under different situations allowing to acquire a more wide range of knowledge than using only one method.

Our proposal is to integrate Knowledge Modelling and Machine Learning moreover to integrate Machine Learning with Problem Solving. In this chapter we provide a summary of Knowledge Modelling methodologies and Machine Learning methods in order to detect how both topics could be integrated. We also analyse some systems integrating Knowledge Acquisition and Problem Solving and some Multistrategy Learning Systems. Both kinds of systems are interesting specially when we want to deal with complex problems. So, in this chapter we revise some concepts that will be useful to achieve our goal.

In the next sections we review the most common Knowledge Acquisition techniques specially focusing on knowledge modelling. Then, in section 3 Machine Learning techniques are explained. In section 4 we explain how the integration between Knowledge Acquisition and Problem Solving can be made. Section 5 explains Multistrategy Learning Systems that combine several basic learning techniques to refine KB. Finally, in section 6 we explain how our work is included in this context and which are the goals that we want to achieve.

2. Knowledge Modelling

Knowledge Acquisition is the process of obtaining the necessary knowledge to develop a KBS. In the KBSs built during the seventies, knowledge was acquired by means of a knowledge engineering phase. Knowledge engineering is a hard process in which a domain expert and a knowledge engineer have to agree on the vocabulary to be used and then the knowledge engineer has to formalise the knowledge provided by the domain expert into a knowledge representation language. Resulting KBSs are domain-specific in the sense that they do not have reusable parts. Several techniques to improve the knowledge acquisition process have been proposed. One of these proposals was to apply the knowledge level framework (Newell, 1982) to the knowledge acquisition process. The adoption of a knowledge level framework focuses the analysis of expertise on issues such as what a task requires and the kind of model that the expert makes of the domain. Methodologies based on a knowledge level analysis are called *model-driven acquisition* or *knowledge modelling* (KM) methodologies and they focus on constructing the models of the problem solving behaviour.

$$\begin{tikzcd}
\text{Abstracted data} \arrow[r, "\textit{heuristic match}"] & \text{Abstracted solution} \arrow[d, "\textit{solution refinement}"] \\
\text{Data} \arrow[u, "\textit{data abstraction}"'] & \text{Solution}
\end{tikzcd}$$

Figure 2.1. Heuristic classification inference structure.

The first step to explicitly structure the knowledge according to the role played in problem solving was the modelling of the heuristic classification systems made by Clancey (1985). Figure 2.1. shows a general, domain-independent analysis of classification systems. Heuristic classification systems have a known finite set of solutions and the solution space is formed by hierarchically organised classes. The reasoning scheme is bottom-up from data to more general knowledge, allowing the inference of some plausible solution classes. In the solutions, the process is top-down by generating more specific solutions compatible with the plausible classes previously inferred.

This model of heuristic classification is not completely satisfactory mainly due to its generality, i.e. most expert systems can be analysed in terms of it. McDermott (1988) developed several knowledge acquisition tools that emphasised the problem solving method as the central key in understanding and building an application. Using this approach is more easy to determine the domain knowledge that the expert has to provide, i.e. the necessary domain models to achieve a goal.

Currently, knowledge engineering is considered as an activity of modelling or constructing models, for which methodologies as Generic Tasks (Chandrasekaran, 1986), KADS (Wielinga et al., 1992) and Components of Expertise (Steels, 1990) have been developed. All these methodologies use concepts such as goals to achieve, knowledge necessary to achieve a goal and different ways to achieve a goal. Moreover, the knowledge modelling methodologies allow the construction of libraries of conceptual models that guide the knowledge acquisition process. In the following sections these methodologies are reviewed. An analysis of how the knowledge level can be useful in the KBS development can be found in (Van de Velde, 1993).

2.1. Generic Tasks Framework

The analysis of the expertise in terms of tasks has shown that some tasks are shared by many KBS, for instance classification or diagnosis tasks. Chandrasekaran (1986) has called this framework *Generic Tasks*. Typical generic tasks are classification, interpretation, diagnosis and construction (planning and design). A generic task can be decomposed in subtasks which can be also shared by many KBS.

A main consequence of the generic tasks framework is that they also use the same kind of both models and inferences independently of the application domain. Focusing on tasks and task decomposition is important because, from a theoretical point of view, it provides a way to build a theory of expertise that makes significant empirical generalisations. This theory should identify a set of generic tasks and give, for each one, what kind of problem solving methods and what kind of domain models are expected. Once this theory has been constructed, strong models for interpreting knowledge acquisition data are available.

The Generic Tasks framework developed by Chandrasekaran (1986) is characterised by three main properties: 1) the kind of problem that solve, 2) how the knowledge has to be represented in order to solve a task, and 3) the control strategy to be used to solve a task. The language and the strategy associated to a task characterise the kind of knowledge to acquire, the decomposition of a problem in sub-problems, and how the system can be implemented. Thus, a generic task can be viewed as a pattern that solves general problems in different domains and that can be reused if both the representation and the inference process are respected. The identification of generic tasks is a way to start systematically cataloguing domain models and problem solving methods. The generic tasks have the following problems: 1) how to decide which tasks are the generic ones and what is their appropriate granularity, 2) the ambiguity between tasks and problem solving methods, 3) the rigidity in the task-method association, and 4) sometimes is not possible to establish a correspondence between a kind of problem and a kind of control strategy.

2.2. KADS

The origin of the KADS methodology was an European project whose goal was to develop a methodology for supporting all the phases of the KBS development. In particular, to support the knowledge acquisition phase, KADS (Wielinga et al., 1992) defined the *model of expertise* composed of four layers (figure 2.2): domain layer, inference layer, task layer and strategic layer. The *domain layer* contains static knowledge (as domain concepts), relations and complex structures as models of processes or device). The *inference layer* contains the description of inferences without specifying how or when they are applied. Elements at this level are knowledge sources, metaclasses, and dependencies between them. A *knowledge source* identifies a kind of inference, i.e. an operation that applied on an input state produces a different output state. So, inferences can be viewed as operations on concepts and knowledge sources can be viewed as abstract operations on abstract concepts.

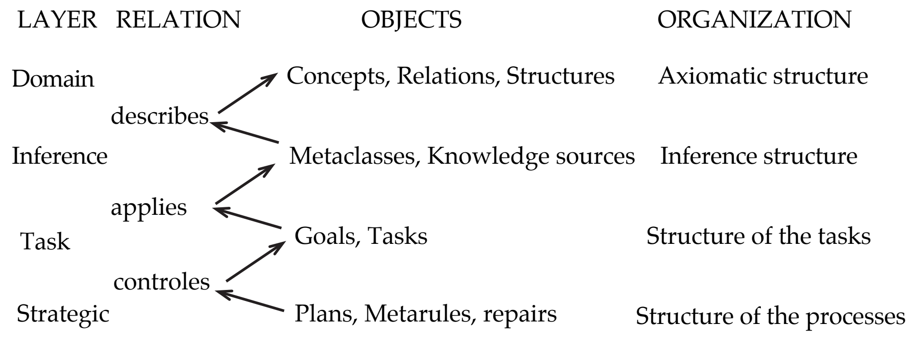


Figure 2.2. Layers of the expertise according the KADS methodology.

Elements used by the *task layer* are goals and tasks. *Tasks* are different ways in which knowledge sources can be combined to obtain a goal, i.e. a task is a problem solving action representing a fixed domain-independent strategy to achieve goals. The task definition consists of the task goal, the input and the output, and the relation between them. In the task body there is a description of how the goal can be achieved by decomposing the task in subtasks. A *task structure* is a fixed strategy controlling the use of the knowledge sources and the user interactions. Finally, the *strategic layer* represents the control capability over the task layer of the problem solving. The knowledge contained in this layer is often represented as rules associating to each kind of problem a particular task structure. The more complete and detailed is the specification of the strategic layer, the more flexible will be the KBS obtained.

The knowledge acquisition process proposed by the KADS methodology is composed of three phases: pre-analysis, analysis and design. The goal of the *pre-analysis* phase is to define relations and concepts of the application domain. There are two sub-phases: the *identification* sub-phase, that obtains objects and domain concepts and establishes a dictionary, and the *conceptualisation* sub-phase in which individual objects are grouped in conceptual primitives (classes and relations). The *analysis* phase is made in two steps: the epistemological analysis and the logic analysis. During the epistemological analysis, the structural properties of each concept are described. During the logic analysis each structure of an expertise layer is formalised. The result of this phase is a set of partial specifications describing how to represent the application domain using the chosen model.

KADS defines interpretation models for tasks (i.e. diagnosis, assessment, monitoring, prediction or design) that are generic patterns guiding the knowledge acquisition process and establishing the decomposition of tasks in subtasks. Interpretation models are independent of the application domain but their application range turns out to be small. That is because the selection of an interpretation model implies the use of all its subtasks. In other words, if some subtask composing an interpretation model is not used, the interpretation model is not useful and another one has to be defined. The knowledge of a KBS can be described by means of a structure of tasks, i.e. a tree representing the decomposition of tasks in subtasks.

The KADS methodology has been widely accepted by many enterprises and research centres mainly due to its available library of interpretation models. CommonKADS (Wielinga et al., 1993) is an improvement of the KADS methodology that supports more aspects of the KBS development, as project management, organisational analysis, knowledge acquisition, conceptual modelling, user interaction, system integration and design. CommonKADS itself does not provide comprehensive support for the whole process of knowledge acquisition and expertise modelling. Instead, the support focuses on the model-driven approach. In addition to the library of generic models, there are also guidelines on how to do the work.

2.3 Components of Expertise

The Components of Expertise (Steels, 1990) is a framework for describing the expertise at the knowledge level. This framework supports the three main activities of the KBS construction, such as knowledge analysis, knowledge acquisition and knowledge coding. The main idea is the decomposition of the expertise in three perspectives: task perspective, model perspective and method perspective. The *task perspective* is described in terms of components as tasks and task structures. In the *model perspective* a characterisation of models and their dependencies is obtained. Finally, in the *method perspective*, methods and control diagrams describing operations of models are identified. Thus, for each perspective a different structure is introduced, task structure, dependency diagram of models and control diagram respectively. In the following sections these three perspectives are analysed.

2.3.1. Task perspective

A *task* is a set of goals that a problem solver has to achieve. Usually, a domain application can be represented by a task that can be decomposed in subtasks, which in turn, may be successively decomposed in subtasks until elemental subtasks are obtained. This decomposition produces a task structure (see figure 2.3) that may be different according to the problem solver. The task structure is represented by a tree, where each node is a task and the children nodes are the subtasks in which this task is decomposed. This is an *and/or* tree because it may not be necessary to solve all the subtasks. The task structure does not specify the order in which the subtasks have to be executed, instead the order in which it is explored is provided by a so called *control diagram.*

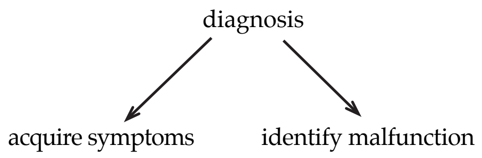


Figure 2.3. Decomposition of diagnosis task (adapted from Steels, 1993).

Using this framework several generic tasks have been detected (diagnosis, description, selection, planning or classification). There have been also several attempts to construct a taxonomy of generic tasks but a consensus about the standard terminology has not been achieved.

2.3.2. Model perspective

The problem solving process can be viewed as a modelling activity that builds a model of several relevant aspects of the reality to find a problem solution. The model perspective is focused on different models that the problem solver can use. Three kinds of models can be distinguished: *case models* that are models of a concrete situation; *domain models* containing domain knowledge that can be useful for the problem solving; and *problem solving process models* that are models of the problem solving process itself. Models and their dependencies are very important, and they are represented in the model dependency diagram (see figure 2.4). From the knowledge level point of view, problem solving methods are processes organising and executing the activities of model construction.

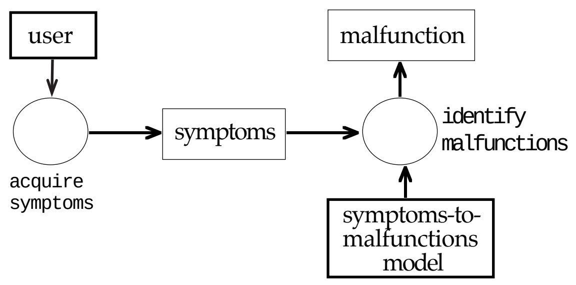


Figure 2.4. Model dependency diagram for diagnosis task (adapted from Steels, 1993).

The analysis of models can be made in two steps: the first one is to identify the different case models and their dependencies; and the second step is to identify domain models determining the role that they play in the construction of case models.

Depending on the user's programming style, a different number of case models can be identified. Usually, a case model is introduced when a problem solving activity focuses on it. Some typical case models are the following: components model, descriptive model, classificatory model, temporal model, spatial model, causal model or functional model. Each case model also has a set of primitive elements, for example, the descriptive model has characteristics, the classificatory model has classes, and the temporal model has temporal relations. Moreover, elements of a model can be organised in a hierarchical structure. To identify case models, the knowledge engineer needs the knowledge provided by a domain expert. Once the case models have been identified, their dependencies have to be determined, i.e. the model dependency diagram has to be constructed.

Domain models are valid models for several cases since they contain domain-specific knowledge necessary to construct case models. The dependency diagram is completed by adding domain models. There are two classes of domain models: expansion models and mapping models. *Expansion* models contain relevant knowledge to expand a model (for example, to add symptoms to a symptom case model). Typical expansion models are descriptive models, default models, or hierarchies that can be used to refine facts of a case model. *Mapping* models are used to construct or to modify a set of case models according to a mapping of elements belonging to other case models (for example from symptoms to malfunctions, or from malfunctions plus symptoms to states of components). Typical mapping models are description-to-class models, function-to-component models, or symptom-to-malfunction model.

2.3.3. Method perspective

A problem solving method (PSM) is a process organising and executing the steps from which several case models can be constructed. The method perspective focuses on how and when the knowledge is used. On the one hand, a method needs a mapping from tasks to models. Methods have to impose a control structure about the order in which tasks have to be executed. Both informations are represented by *control diagrams*. These diagrams are graphs where nodes are tasks and links between tasks occur when the goal of a task has been achieved (see figure 2.5). Labels of links are conditions that have to be fulfilled to activate a new task.

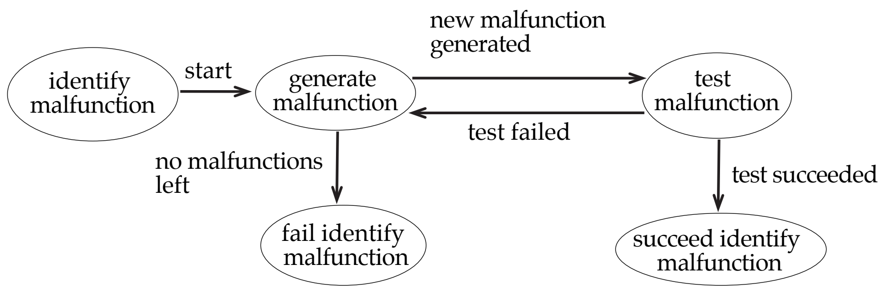


Figure 2.5. Control diagram for diagnosis (adapted from Steels, 1993).

Methods can belong to three main classes: task decomposition methods, task execution methods and search methods. *Task decomposition* methods neither add information to case models nor consult domain models. Typical task decomposition methods are the divide-and-conquer method, the progressive-refinement method and the propose-and-revise method. Task decomposition methods can be divided in turn in two kinds of methods: mapping methods and expansion methods. Mapping methods use one or more case models to build or modify a set of case models. Expansion models focus on the development of a case model by adding new knowledge to them.

Task execution methods split up a task into subtasks whose execution results in problem solving activities that build case models. A typical example of task execution method is linear classification. *Search* methods are necessary when there is not enough information to decide how a case model has to be expanded. This situation arises either when the mapping model is incomplete, or when the expansion model is incomplete. In this situation the problem solver has to explore different alternatives to select the most appropriate of them.

2.4. PSM libraries

The need to increase modularity has been addressed by the construction of *libraries* of components. These libraries are formed by generic components which may be used in a domain-independent way. The construction of these libraries has been specially useful because of task decomposition methods. KADS already proposed the construction of libraries of interpretation models, but they are not flexible enough since all tasks composing a method must be used. Some problems derived of the lack of flexibility of the KADS library have been analysed by Cañamero (1995) and Orsvärn (1996). Generic Tasks and Components of Expertise allow a higher reusability and modularity of problem solving methods (PSM). In these methodologies, the knowledge acquisition is a top-down activity selecting methods in a task/subtask decomposition.

Task decomposition methods have a small granularity; therefore they have a high possibility of reuse, i.e. they are more generic. Nevertheless, the identification of generic task decomposition methods is not easy since an analysis of requirements is necessary in order to assure a real possibility of reuse. The analysis of requirements defines the *applicability problem*, i.e. to define under which conditions a PSM can be applied to solve a task. Applicability conditions are determined in the knowledge engineering phase and may be used to dynamically decide the PSM that can be used. Benjamins and Pierret-Golbreich (1996) have called *assumptions* the applicability conditions of methods. They have organised assumptions in two forms: *horizontally* according to a specialisation hierarchy; and *vertically* allowing determine the completeness, consistency and redundancy of assumptions. Thus, it is necessary to known the assumptions associated to a PSM to be able to reuse it.

Once problem solving methods and their corresponding assumptions have been defined, two problems appear (Studer et al., 1996): the *indexing problem* and the *configuration problem*. The indexing problem consists in the location of an appropriate PSM into the library. The configuration problem consists of how a PSM can be adapted to solve the current task. The tool PROTÉGÉ-II (Puerta et al., 1992) is specially addressed to the configuration problem. The configuration problem is also described in (Albert and Rouveirol, 1994) although the description of assumptions of the different methods used in this work was rather informal.

2.4.1. Selection of PSM

Knowledge modelling methodologies allow constructing KBS in two independent phases: the design phase and the implementation phase. The design phase has the following goals: 1) to acquire all the necessary knowledge, 2) to determine which tasks have to be solved, and 3) to select a method to solve each task. The selection of an appropriate method to solve a task is an issue that influences the flexibility of a KBS. In other words, the KBS strategy remains fixed once a PSM is chosen for each task. The main problem that can appear is that the chosen PSM uses some non available knowledge, therefore the task cannot be solved. A solution to this problem could be to solve the task using a different PSM, i.e. make a flexible selection of the PSM.

Several authors have addressed their research to build more flexible KBS. Most of them (Chandrasekaran, 1990; Wanwelkenhuysen and Rademakers, 1990; Van Marcke, 1990) use knowledge modelling methodologies, representing the problem solving in terms of models, tasks and methods. A task can be solved using several methods, which have applicability conditions that are evaluated during runtime. Benjamins and Pierret-Golbreich (1996) make a classification of the applicability conditions of methods (that they call *assumptions*) and propose a metalevel to estimate the utility of each method and select the most appropriate.

Chandrasekaran (1990) provides a theoretical framework of how a flexible selection in run-time could be made. Wanwelkenhuysen and Rademakers (1990) propose a computational framework (implemented on top of the KRS language (Van Marcke, 1988)) which attempts to closely connect entities at the knowledge level with objects at the implementation level. An appropriate method to solve a task is dynamically made among those methods whose applicability conditions are satisfied. Van Marcke (1990) defines a Generic Tutoring Environment (GTE) in which teaching expertise is represented in terms of instructional tasks, instructional methods, and instructional primitives. Tasks in GTE can be solved using several methods, each representing an alternative way to perform the task. Applicability conditions of the methods in GTE are numbers. The method having the highest applicability number is selected as the most appropriate to solve a task. The applicability number of a method is computed according to a function involving the current context and state and the different sources that the current method has used.

In chapter 3 we describe a framework integrating KM and ML in the development of KBS applications. In this framework we propose:

1) to acquire knowledge using both KM and ML,

2) to integrate learning methods and PSM, and

3) to allow a lazy selection of a PSM for a task.

In this framework, the integration of Machine Learning and Problem Solving Methods is based on Knowledge Modelling. Thus, from the Knowledge Modelling of a domain we determine the set of tasks that have to be solved. When a task requires some knowledge, it may be acquired during the design phase or delayed until execution time. In our approach, tasks that acquire knowledge, during the design phase or at execution time, can use ML methods.

In our approach, the selection of a PSM for a task can be delayed until runtime. To achieve the dynamic selection of one PSM for a task, we propose a lazy problem-centred approach. This approach consists of delaying some decisions until they are needed (i.e. at execution time) in order to use the concrete information available about a particular problem. Using the lazy problem-centred approach we can select the appropriate PSM for a task according to the specific problem to solve. Thus, during the design phase, more than one PSM can be associated to a task in the application domain using different knowledge resources. The KM process has to establish the conditions under which each PSM can be applied and/or the preferences in the application of each PSM if more than one is applicable. The selection of the appropriate PSM for each task is delayed until a specific problem has to be solved. In that moment the KBS application can take into account both the applicability conditions and the preferences with respect to the particular problem or situation being addressed.

3. Machine Learning

Learning is the ability exhibited by humans to adapt and modify its behaviour according to experience. Machine Learning has as goal to analyse and model learning processes in all its multiple manifestations.

Machine Learning research has been focused in three main topics (Carbonell et al., 1983): the development and analysis of learning systems improving the performance of a predetermined set of tasks, the research and simulation of human learning processes and the theoretic research of methods and algorithms applicable in a domain-independent way.

Early learning programs (non-symbolic) were applied to adaptive control systems. Symbolic learning appears when learning results need to be understood by human users. The symbolic paradigm uses logic or graphic structures to represent the knowledge and learn symbolic descriptions representing high-level knowledge.

Learning may be used as a way to acquire knowledge or associated to a concret problem solving system. *Inductive learning methods* (analysed in next section) are typically used to acquire general knowledge from examples. *Lazy methods* are those in which the experience is accessed, selected and used in a problem-centred way. In section 3.2. lazy methods such as instance-based and case-based reasoning are analysed.

3.1. Inductive Learning

Inductive methods can be applied in two ways: as interactive tools acquiring knowledge from examples or as a part of learning systems. When inductive methods are used to acquire knowledge, the user provides examples and strongly controls the use of the method. Used as part of learning systems, inductive methods are activated when another system component has the necessity to learn from positive and/or negative examples that constitute the feedback from which the system can achieve the current task. Example of systems having inductive learning integrated with problem solving are LEX (Utgoff, 1986).

Induction is based on specific facts (examples) instead of general axioms as in deduction. The goal of the induction is to formulate plausible general assertions that both explain the given facts and are capable to predict unseen facts. In other words, the inductive inference tries to obtain a complete and correct description of a given phenomenon from specific (maybe partial) observations of it. The description inductively obtained is true at least for the already seen examples but nothing can be assured for unseen examples.

The most frequent application of inductive learning is *concept learning*. The goal of concept learning is to find symbolic descriptions expressed in high-level terms that are understandable by people. Concept learning can be defined as follows:

Given: A set of (positive and negative) examples of a concept and eventually some background knowledge

Find: A general description (hypothesis) of the concept describing all the positive examples and none of the negative examples.

Background knowledge defines assumptions and constraints imposed on examples and generated descriptions, and any relevant domain knowledge. Background knowledge can be in different forms (Michalski, 1983), e.g. in declarative form or in procedural form, as sequences of instructions for executing specific tasks (control knowledge).

Concept learning can be viewed as the task of searching through a large space of hypotheses implicitly defined by the hypothesis representation language (Mitchell, 1992). The goal of this search is to find the hypothesis (description) that best explains the examples. The language used is very important since it defines the hypothesis space, i.e. what knowledge can be expressed, and therefore what knowledge can be learned. The language has to be chosen with care in order to easily express all the desired knowledge. Most commonly used languages are constrained forms of predicate calculus, like decision trees, production rules, semantic nets and frames.

The background knowledge includes a preference criterion allowing the reduction of the set of hypotheses to a smaller one containing the most preferable hypotheses. Typically, the preference criterion characterises the desired properties of the searched inductive hypothesis. This criterion is necessary when the description language is complete, i.e. all possible hypotheses can be expressed. An alternative way to constraint the hypothesis space is using a biased description language in which not all the possible hypotheses can be expressed (i.e. the language is incomplete).

Utgoff and Mitchell (1982) defined *bias* as anything influencing the way in which the induction is made. Thus, the notion of bias includes any input besides examples, and any parameter or strategy that may be modified by the user of a learning system. The declarative bias, i.e. a bias explicitly given by the user, has as advantage the possibility to be used in several systems and allows meta-level reasoning about it. This second advantage is important since if a current bias is considered insufficient, it may be changed (Utgoff, 1986). In (Nedellec et al., 1996) a complete analysis of declarative bias can be found.

Inductive concept learning methods can be classified according to the following perspectives:

- *Supervised/unsupervised.* Supervised methods need an oracle that provides the classes (concepts) to which the examples belong and classifies the examples. Unsupervised methods have to discover the concepts to which the examples belong. In our work have focused on supervised methods, i.e. we will suppose that the oracle has provided the concepts and has classified the examples.

- *Single/multiple concept learning.* This classification is according to the number of concepts that have to be learned. Single concept learning can be achieved in two situations: 1) inputs are only positive examples, 2) inputs are positive and negative examples. Multiple concept learning also distinguishes two cases: 1) when the descriptions of the different classes (concepts) are mutually disjoint, 2) when the descriptions of concepts overlap, i.e. an example can satisfy the descriptions of several classes. Multiple concept learning has been implemented in AQVAL/1 and AQ11 (Michalski and Larson, 1978). Learning multiple concepts raises the problem of learning dependent predicates (including the case of mutually dependent predicates. In (De Raedt et al., 1993) and (Bergadano and Gunetti, 1993) can be found an analysis of these problems.

- *Propositional/relational learners.* Methods using a formalism equivalent to propositional calculus are called *attribute-value learners* or *propositional learners.* These methods use objects described as a fixed collection of attributes, each of them taking a value from a corresponding pre-specified set of values. Methods that learn first-order relational descriptions are called *relational learners.* They induce descriptions of relations and use objects described in terms of their components and relations among components. The background knowledge is formed by relations which, as the language of examples and concept descriptions, are typically subsets of first order logic. In particular, learners that induce hypotheses in the form of logic programs (Horn clauses) are called *inductive logic programming* systems.

An important application of inductive learning is the automatic construction of KB for Expert Systems, being an alternative to classic knowledge acquisition methods. Inductive techniques can also be used for the refinement of existing KB since they allow the detection of inconsistencies, redundancies, and lack of knowledge, and also allow the simplification of the rules provided by the domain expert. Application domains such as biology, psychology, medicine and genetics take benefit from the capability of inductive methods to detect patterns present in the input examples. In the following sections both propositional and relational learners are explained in more detail.

3.1.1. Propositional Learners

Propositional learners use examples described in terms of a fixed set of attributes, each one having its own set of values. Examples are classified, i.e. the class to which they belong is known. The abstraction level of attributes affects the induced description, since high-level attributes produce more understandable descriptions, i.e. more compact descriptions. Thus, the selection of attributes describing the examples determines the scope of the descriptions that can be learned (representational bias). Propositional learners assume that all the examples are described using the same set of attributes, otherwise they need mechanisms for handling imperfect data. The learning task of propositional learners can be described as follows:

Given: a set of correctly classified training examples

Find: a rule predicting the class to which seen or unseen examples belong.

Representative propositional learners are the family of AQ algorithms (Michalski, 1983) and the ID3 algorithm (Quinlan, 1986). AQ algorithms induce description rules (having an if-then form) for each class. A class is described by a disjunctive logic expression, i.e. the disjunction of several clauses. The description of each class (hypothesis) is searched taking as negative examples the training examples belonging to other classes and applying the *set covering* algorithm explained in section 3.1.2.1.

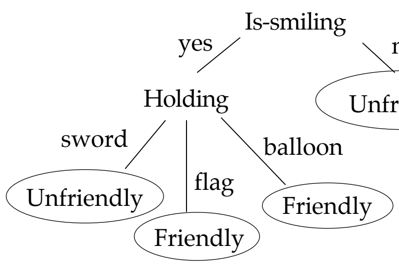


Figure 2.6. An example of decision tree belonging to robots domain (Lavrac and Dzeroski, 1994). The set of values for the is-smiling attribute is {yes, no}, and the set of values for the attribute holding is {sword, flag, balloon}.

The ID3 algorithm expresses the learned knowledge using *decision trees.* Each internal node of a decision tree is labelled by an attribute and links from a node are labelled by the possible values of the attribute (see figure 2.6). The tree construction process is based on the selection of the most informative attribute, trying to minimise the number of tests (i.e. the length of paths from root to leaves). The algorithm used by ID3 to build a decision tree is the following (from Lavrac and Dzeroski, 1994):

```
Let E the set of examples and $C_1$ … $C_p$ the set of classes in which the examples
can be classified

Initialise $E_{curr}$:= E; T:= nil

function DT(T,$E_{curr}$)
  if all examples in $E_{curr}$ belong to the same class $C_s$
     then generate a leaf labelled $C_s$
     else {generation of a new node}
          select the most informative attribute A with values {$v_1$ … $v_n$} for the
                root of the tree
          Split $E_{curr}$ into subsets $E_1$ … $E_n$ according to the values $v_1$ … $v_n$
          for i = 1 to n do
             DT($T_i$,$E_i$)          ;; recursively build a subtree $T_i$ for $E_i$
          end-for
  end-if

output: Decision tree T with the root A and subtrees $T_1$ … $T_n$
```

If all the input examples in E_{curr} belong to the same class C (represent the same concept), the decision tree contains only a leaf corresponding to C. Otherwise, input examples belong to several classes C_1 … C_n. In that situation, the algorithm selects an attribute and divides E in disjoint sets E_1 … E_n. Each set E_i of the partition contains examples having the same value in the selected attribute. The algorithm is applied to each partition set E_i until the obtained sets contain elements belonging to a unique class. Tree leaves are classes C_1 … C_n in which the examples can be classified.

An unseen example is classified starting from the root, testing attributes of internal nodes, to a leaf. The assumption made in the decision tree is that examples belonging to different classes have different values at least one of their attributes.

The selection of the attribute that divides the set of examples is a crucial decision. There are many possible measures to make this selection, in particular the entropy measures and the Gini's index are the most commonly used. The main idea is to select an attribute producing a maximum information gain. ID3 uses the following expression to compute the gain obtained in selecting the attribute A_k

$$\text{Gain } (A_k, X) = I(X) - E\ (A_k, X)$$

where $I(X) = -\sum_{j=1}^{m} P_j \log_2 P_j$ with $P_j = \frac{|X \cap F_j|}{|X|}$ and $E(A_k, X) = \sum_{i=1}^{n} \frac{|X_i|}{|X|} I(X_i)$ being m the number of possible classes, n the number of possible values of the attribute A_k, $|X|$ the number of examples in the node and $|X_i|$ the number of examples in X having the value v_i in the attribute A_k. I(X) measures the randomness of the distribution of the examples in X over m possible classes. P_j is the probability of occurrence of each class F_j in the set of examples, defined as the proportion of examples in X belonging to the class F_j.

The gain is computed for all the attributes describing the examples, and the attribute having maximum gain is selected. This measure preferentially selects attributes with a large set of values, for this reason Quinlan (1986) introduced a correction defining the Gain Ratio as follows:

$$G_R(A_k,X) = \frac{I(X) - E(A_k,X)}{IV(A_k)} \quad \text{where } IV(A_k) = -\sum_{i=1}^{n} \frac{|X_i|}{|X|} \log_2 \frac{|X_i|}{|X|}$$

This nrmalisation has, however a rather ad-hoc justification. An alternative to the Gain Ratio is the distance-based measure introduced in (López de Mántaras, 1991). This measure is based on defining a distance between partitions of the data. Each attribute is evaluated based on the distance between the data partition it creates, and the correct partition (i.e. the partition that perfectly classifies the training data). The attribute whose partition is closest to the correct one is chosen. It is formally proved (see López de Mántaras, 1991) that such distance measure is not biased towards attributes with a large number of values. Furthermore it avoids the practical difficulties associated with the Gain Ratio and produces statistically significant smaller trees.

In (Kononenko, 1995) an analysis of the behaviour showed by measures commonly used to divide the training set can be found.

Problems exhibited by decision trees can be classified in two categories: algorithmic problems and problems inherent to the representation. The cost of top-down decision tree induction algorithms can be reduced by implementing a greedy approach searching for a small tree. Then, selection measures can be used to estimate which attribute provides the maximum information gain if it is included in the decision tree. Several algorithms have been developed to improve the decision tree when domain concepts are hard (concepts represented by many relevant attributes with high interaction among them). For example, FRINGE (Pagallo and Haussler, 1990) uses the decision tree produced by a greedy splitter to construct new features improving the tree quality. The Lookahead Feature Construction algorithm (LFC) (Ragavan and Rendell, 1994) is a global search algorithm that caches search information as newly constructed features.

Two problems inherent to the decision tree representation are replication and fragmentation (Pagallo and Haussler, 1990). The *replication* problem produces the duplication of sub-trees in disjunctive concepts. Replication degrades accuracy, consistency and comprehensibility. The *fragmentation* problem causes the partition of examples in small sets. Replication always implies fragmentation, but fragmentation can occur without replication if many attributes are tested (long paths). Another problem inherent to the representation is the handling of unknown values. When an attribute of an internal node has an unknown value in the current example, it is not possible to decide how the remaining sub-tree has to be explored. Usually, algorithms have special (and expensive) mechanisms to deal with unknown values.

The main limitations of propositional learners are the limited expressiveness of the representational formalism and their limited capability of taking into account the available background knowledge. The predictivity of propositional learners is highly dependent on the form in which the training set is divided by the heuristics used.

Some systems expressing the learned knowledge as decision trees are CART (Breiman et al., 1984) and ASSISTANT (Cestnik et al., 1987). ASSISTANT extends the ID3 algorithm allowing the manipulation of incompletely specified examples, the binarisation of continuos attributes, the construction of binary trees, the pruning of the tree and the plausible classification based on the naive Bayesian principle.

3.1.2. Relational Learners

Relational learners are able to deal with structured objects, i.e. objects described structurally in terms of their components and relations among components. Learned knowledge are descriptions of relations (i.e. definitions of predicates). In relational learners the languages used to represent examples, background knowledge and concept descriptions are usually subsets of first-order logic.

In the next section we describe FOIL, a system that learns Horn clauses from data expressed as relations. FOIL is based on ideas that have proved to be effective in propositional learners, and extends them to a first-order formalism. In section 3.1.2.2 we introduce a group of relational learners, called *inductive logic programming* systems, that uses knowledge represented as Horn clauses. Many authors consider FOIL as one of the early systems that can be included in the Inductive Logic Programming framework.

3.1.2.1. FOIL

FOIL (Quinlan, 1990) is a system incorporating ideas of both propositional and relational learners. Objects are described using relations from which FOIL generates classification rules expressed in a restricted form of first-order logic. In particular, FOIL uses the set covering approach as in AQ (Michalski, 1983), a heuristic information-based search taken from ID3

(Quinlan, 1986) and the idea of the top-down searching of refinement graphs taken from MIS (Shapiro, 1983).

FOIL follows three main steps: 1) pre-processing of the training set, 2) construction of hypotheses, and 3) post-processing of hypotheses. During pre-processing, negative examples are generated using the closed-world assumption. Post-processing is a pruning process in order to reduce the complexity of the constructed hypothesis. Basically, pruning consists of removing irrelevant literals from a clause and removing irrelevant clauses from the hypothesis (see Lavrac and Dzeroski, 1994).

Hypothesis construction is made using the *set covering algorithm.* Let us suppose that we want to find the description of N classes $C_1 \ldots C_N$. This problem can be transformed in N problems each one finding the description of one class C_i. In this transformation, examples belonging to the class C_i form the set of positive examples and examples belonging to other classes are considered negative examples of the class C_i. The set covering algorithm to construct a hypothesis has the following three steps:

-1- Search for a conjunction of conditions satisfying some examples of the class C_i and none of other classes (clause construction step).

-2- Add the obtained conjunction as a disjunction of the hypothesis that is searching for (hypothesis construction step).

-3- Delete from the training set the examples satisfying the obtained conjunction. If the obtained set is not empty the whole process is repeated until all positive examples are covered.

Clauses are constructed using a *beam search* algorithm. This algorithm begins with a clause c containing all the attributes that describe the examples and each attribute has as value the disjunction of all the possible values. The clause c is successively specialised until no negative examples are covered. The specialisation consists of removing values from attributes (there are many possible values to remove). The obtained specialisations are evaluated using a quality criterion. When at each step there are several rules to choose, they are ordered according to several criteria. First, the algorithm prefers those clauses covering as many examples as possible; then it prefers those clauses having a smaller number of attributes; and finally it prefers those clauses having the least total number of values in the internal disjunctions.

From this set covering algorithm several improvements have been developed. For example AQ15 (Michalski et al., 1986) and NEWGEM (in (Lavrac and Dzeroski, 1994) a brief description of this system can be found) that incorporate incrementality and initial hypothesis provided by the user; AQ17 (Wnek and Michalski, 1991) that incorporates constructive induction; CN2 (Clark and Niblett, 1989) combines the ID3 algorithm to handle imperfect data using the same flexible strategy used by AQ.

Imperfect data are handled in FOIL using a stopping criterion based on the encoding length restriction. Thus the construction of a clause stops when no negative examples are covered or when no more bits are available to add more literals to the body hypothesis. The search for more clauses stops when all the positive examples are covered or when no more clauses may be constructed under the encoding length constraint. In (Quinlan, 1990) the application of FOIL to several domains can be found.

3.1.2.2 ILP

Inductive Logic Programming (ILP) has been defined as the intersection of inductive learning and computational logic (Lloyd, 1990) since it uses techniques of both fields. ILP inherits two goals from the inductive learning: 1) to develop tools and techniques to induce hypotheses from observations (or examples), and 2) to synthesise new knowledge from experience (background knowledge). ILP uses computational logic as the representation mechanism of both hypotheses and observations, avoiding the two main limitations of the classic Machine Learning techniques: 1) use of a limited knowledge representation formalism (usually propositional logic), and 2) problems in using background knowledge. Thus, from computational logic, ILP inherits the representational formalism, its semantics and several well-stablished techniques. For an overview of ILP see (Muggleton and De Raedt, 1994).

The main concerns of ILP are inference rule properties, algorithm convergence and computational complexity of the procedures. ILP extends theory and practice of computational logic using induction as basic inference rule. Plotkin's work on inductive generalisation (Plotkin, 1969), the work on model inference by Shapiro (1983), and the work of Sammut and Banerji (1986) inspired the efforts made by Lapointe and Matwin (1992), and Idestam-Almquist (1993) in studying the implication operator. These authors studied inductive rules regarding them as inverse of deduction rules. This inversion is made by introducing a partial order: the θ-subsumption. Nevertheless, θ-subsumption is not enough to handle recursive clauses since it is incomplete with respect to implication.

ILP research has commonly focused on concept learning, where the examples are implications and the goal is to induce a hypothesis able to classify correctly the examples. Concept learning uses positive and negative examples to induce a discriminant description for a concept. In Data Mining and in Knowledge Discovering in Databases, there is a large amount of data available and the main goal is to find the properties or regularities that they present instead of discriminant descriptions (since all examples are considered positive). Flach (1992) formalised these two types of induction as explanatory and confirmatory induction respectively.

The use of a relational formalisms allows the application of induction over domains in which an attribute-value representation is not sufficient. Thus, ILP is being successfully applied in areas as knowledge acquisition, knowledge discovery in databases, scientific knowledge discovery, logic program synthesis and inductive engineering. Classical ILP applications are protein secondary-structure prediction (Muggleton et al., 1992), finite element mesh design (Dolsak and Muggleton, 1992) and automatic construction of qualitative models (Bratko et al., 1991). An important application is also the construction of programming assistants, that is, tools supporting software design and implementation (Shapiro, 1983; Quinlan, 1990; Kirschenbaum and Sterling, 1991; De Raedt, 1992; Bergadano and Gunetti, 1996).

In the following sections we will describe the basic concepts and techniques of ILP, how the systems using ILP can be classified, and some representative ILP systems.

Basic Concepts

Lavrac and Dzeroski (1994) give the following description of the Inductive Logic Programming (ILP) problem:

Given: Background knowledge BK, and a set of training examples $E = E^- \cup E^+$, where E^- are negative examples and E^+ positive examples

Find: a hypothesis H expressed in some concept description language L, such that H is complete and consistent with respect to BK and E.

Given background knowledge BK, a hypothesis H, and a set of examples E, we say that:

- A hypothesis H *covers* an example $e \in E$ if $BK \cup H \models e$.

According to this definition, *completeness* means that all the positive training examples are covered by H ($BK \cup H \models e_i \; \forall e_i \in E^+$) and *consistency* means that no negative example is covered by the hypothesis H ($BK \cup H \not\models e_i \; \forall e_i \in E^-$).

ILP can be viewed as a search problem (Muggleton and De Raedt, 1994) since there is a candidate solution space and an acceptance criterion characterising the solutions. This is a similar vision of that provided by Mitchell (1982) for concept learning in ML in general. The hypothesis space contains descriptions of concepts and the goal is to find one or more states satisfying some given quality criteria. The space of possible hypotheses is very wide, thus pruning and heuristic techniques are needed to find an appropriate hypothesis. As in concept learning, the hypothesis space in ILP is structured according to the notions of specialisation and generalisation.

- An hypothesis G is *more general than* an hypothesis S if and only if $G \models S$ (we can also say that S is *more specific than* G).

Generalisation corresponds to induction and specialisation corresponds to deduction. Therefore, some authors propose that induction can be viewed as the inverse of deduction. From this assertion both the specialisation and the generalisation rules can be defined as follows (Muggleton and De Raedt, 1994):

- A deductive inference rule is a *specialisation rule* when it maps a conjunction of clauses G into a conjunction of clauses S such that $G \models S$.
- An inductive inference rule is a *generalisation rule* when it maps a conjunction of clauses S into a conjunction of clauses G such that $G \models S$.

The generalisation and specialisation allow pruning the hypothesis space since:

1) if $BK \wedge H \not\models e^+$ no specialisation of H will satisfy e^+
2) if $BK \wedge H \wedge e^- \models \Box$, any generalisation of H will be consistent with $B \wedge e^-$.

The hypothesis space can also be constrained using a bias. The definition of bias used in ILP is the same that Mitchell (1991) introduced. ILP has focused on the analysis of the *declarative bias*, i.e. the bias explicitly provided by the user and that may be a modifiable parameter of the system.

Nedellec et al. (1996) propose three kinds of declarative bias: language bias, search bias and validation bias. The *language bias* defines the language that determines the space of possible descriptions of concepts. The *search bias* determines which part of the hypothesis space is explored and how it is done. Finally, the *validation bias* determines an acceptance criterion, i.e. when the system considers that the searched hypothesis has been found. In (Nedellec et al., 1996) a further analysis of these three kinds of bias can be found.

Declarative bias influences learning in the sense that a strongly biased (less expressive) language produces smaller search space and, thus a more efficient learning (see Vapnik and Chervonenkis, 1981). Strongly biased languages have a shortcoming: sometimes the solution cannot be expressed in this language and therefore, it cannot be found. On the other hand, an expressive language allows more possible hypothesis, therefore the method has to perform more search in order to find the appropriate hypothesis.

Once the hypothesis space has been defined, a systematic procedure to explore it has to be defined. The usual way is to introduce a partial order between clauses. This partial order can be based on the θ-subsumption defined by Plotkin (1969).

- A clause c *θ-subsumes* a clause c' if there exists a substitution θ such that $c\theta \subseteq c'$.
- Two clauses c and c' are *θ-subsumption equivalent* if c θ-subsumes c' and c' θ-subsumes c.

- A clause is *reduced* if it is not θ-subsumption equivalent to any proper subset of itself.

According to this definition a clause c is *at least as general as* a clause c', written $c \leq c'$, if c θ-subsumes c'. A clause c is *more general than* c' if $c \leq c'$ but not $c' \leq c$.

In (Muggleton and De Raedt, 1994) several properties of the θ-subsumption can be found. We want to emphasise two of these properties:

1) if c θ-subsumes c' then $c \models c'$(the inverse is not true, Flach, 1992)

2) θ-subsumption defines a lattice over the set of reduced clauses. This means that there is a unique least upper bound (*lub*) and a unique greatest lower bound (*glb*).

θ-subsumption between clauses is decidible and easy to compute but sometimes does not exist. Nevertheless, using Horn clauses with a common predicate symbol and having the same sign the lgg under θ-subsumption can always be found. Thanks to the lattice formed under the θ-subsumption the least general generalisation can be defined as follows (Lavrac and Dzeroski, 1994):

- The *least general generalisation* (lgg) of two reduced clauses c and c', lgg(c, c') is the least upper bound of c and c' in the θ-subsumption lattice.

The length of the lgg of two clauses C_1 and C_2 is at most $|C_1| \times |C_1|$. A set of literals has a unique lgg, but several lgg can exist for a set of clauses.

Summarising, θ-subsumption is important because allows to structure the hypothesis space and prune its exploration. Also, θ-subsumption serves as basis for two strategies exploring the hypothesis space: bottom-up (or specific to general) strategy and top-down (or general to specific) strategy. Both strategies will be analysed in the following section.

Bottom-up strategy

Bottom-up methods are based on the generalisation of a set of positive examples. The bottom-up strategy proposed by Plotkin consists of building the least general generalisation of the training examples (without using the negative examples). There are three basic generalisation techniques used to generate hypotheses: relative least general generalisation, inverse resolution and inverse implication.

Relative least general generalisation (Plotkin, 1969). The least general generalisation (lgg) generalises the positive examples without using background knowledge. Relative least general generalisation (rlgg) is introduced to take into account the background knowledge. Thus, rlgg is defined from the lgg as follow:

- The *relative least general generalisation* (rlgg) of two clauses c and c' is their least general generalisation lgg (c, c') relative to background knowledge BK.

Plotkin showed that the rlgg of a set of clauses sometimes does not exist, or if exists, the rlgg can have infinite length or it may be hard to compute. Using the ij-determinacy as language bias (Muggleton and Feng, 1990), the construction of a unique and finite rlgg is assured. Buntine (1988) defined a special case of relative subsumption, called *generalised subsumption*, only applicable to definite clauses.

- A clause c is *more general than* a clause c' with respect to a background knowledge BK, if any example e that can be explained using c' and BK ($BK \cup c' \models e$) can also be explained using c ($BK \cup c \models e$).

Generalised subsumption degenerates into Plotkin's θ-subsumption in absence of background knowledge. Thus, θ-subsumption can be considered as a special case of generalised subsumption. On the other hand, generalised subsumption is a special case of rlgg. Generalised subsumption produces a hypothesis space smaller than the one obtained by the θ-subsumption, but it is harder to compute. In order to make easy the computation of generalised subsumption, Buntine introduced the *most specific generalisation* notion. Nevertheless, generalised subsumption has remained only as a theoretic result.

Inverse resolution. Inverse resolution was introduced in first order logic by Muggleton and Buntine (1988). These authors introduced inverse resolution in ILP as a generalisation technique inverting Robinson's resolution rule. The idea is that because the resolution rule is complete for deduction, inverse resolution could be complete for the induction. Muggleton (1987) proposed four inverse resolution rules: absorption, identification, intra-construction and inter-construction. The absorption and the identification rules invert a single resolution step and they are called *V-operators*. The intra-construction and the inter-construction rules are called *W-operators*. In (Bergadano and Gunetti, 1996) more information about these operators can be found. W-operators are used in *predicate invention* (Stahl, 1996), a field (like *constructive induction*[1] in Machine Learning) with the goal of automatically extending the vocabulary used by the system whenever the vocabulary is insufficient to learn the desired concept. In (Ling, 1991) there is an analysis of the conditions under which predicate invention is necessary.

[1] *Constructive induction* (Michalski, 1983) is a research field of Machine Learning studying the problems that appear when the desired goal cannot be learned due to the bias at the vocabulary level. Induction is *constructive* when it generates descriptors different from those used to describe input facts. Some main issues are to determine when a new descriptor is necessary, when the new descriptors can be applied and how to evaluate their quality. INDUCE-I (Larson and Michalski, 1977) is one of the early systems using constructive induction.

Inverse implication. Since implication is undecidible (Schmidt-Schauss, 1988), implication inversion may be very expensive. Nevertheless, the implication inversion can be made using restrictions that limit the kind of clauses than can be learned. Implication inversion is interesting to induce self-recursive clauses. There are three approaches to invert implication: searching structural regularities of terms, finding structural regularities of literals and finding internal and external connections. These approaches are explained in depth in (Bergadano and Gunetti, 1996).

Some systems using a bottom-up strategy are the following: ITOU (Rouveirol, 1992), CLINT (De Raedt, 1992), MARVIN (Sammut and Banerji, 1986), and GOLEM (Dolsak and Muggleton, 1992).

Top-down strategy

Top-down strategy is achieved by means of specialisation techniques. Specialisation techniques search in the hypothesis space from general to specific using specialisation operators, also called *refinement operators* (Shapiro, 1983), based on θ-subsumption. A refinement operator can be defined as follows (from Lavrac and Dzeroski, 1994):

- Given a language bias L, a *refinement operator* ρ maps a clause c to a set of clauses $\rho(c) = \{c' \mid c' \in L, c < c'\}$ which are specialisations (refinements) of c.

A refinement operator performs two operations: 1) applies a substitution θ to a clause, and 2) adds a literal (or a set of literals) to a clause. The properties of the refinement operators (i.e. global completeness, local completeness and optimality) can be found in (Muggleton and De Raedt, 1994). These properties are desirable to assure both that all the possible hypotheses are considered (global and local completeness) and that each clause is considered only once (optimality). A generic top-down algorithm is the following (from Lavrac and Dzeroski, 1994):

```
Let E be the set of examples and H the hypothesis under construction
Initialise $E_{curr}$ := E; H := Ø

repeat {covering}
  Initialise c := T ← .
  repeat {specialisation}
    Find the best refinement $c_{best} \in \rho(c)$
    Assign c := $c_{best}$
  until Necessity stopping criterion is satisfied
  H' := H ∪ {c}    ;; Add the clause c to H to get new hypothesis
  $E'_{curr}$:= $E_{curr}$-covers(B,H',$E^{+}_{curr}$) ;; Remove positive examples covered
                                     ;; by c from $E_{curr}$
  Let $E_{curr}$ := $E'_{curr}$ and H := H'

until sufficiency stopping criterion is satisfied
```

The top-down algorithm needs the whole training set (containing positive and negative examples). The construction of a hypothesis starts from the most general clause (the empty clause) and repeatedly refines (specialises) it until no negative examples are covered. The refinement of a clause is made by adding a new literal to the existing clause. A hill-climbing strategy can be used to obtain the best refinement, which is then taken as the new clause to refine. This process is repeated until a clause satisfying the stopping criteria is found. The necessity criterion decides when it is not necessary to add more literals to a clause. The sufficiency criterion decides when is not necessary to add more clauses to an hypothesis. Several stopping criteria can be used to decide whether the domain data are perfect or not. If data is perfect, i.e, without errors, necessity and sufficiency stopping criteria are, respectively consistence (no negative examples are covered) and sufficiency (all the positive examples are covered). Data is imperfect when some kind of (random or systematic) error is present. Most usual errors are the following (Lavrac i Dzeroski, 1994):

- noise, random errors in the examples and/or in the background knowledge
- the problem space is insufficiently covered, i.e. the known examples are not a good sample of the problem space
- inaccuracy, the description language is not sufficient or is inappropriate to express the exact description of the concept to learn
- unknown values in the examples

Noise, inadequate samples and inaccuracy are usually solved by relaxing the completeness and consistency criteria, allowing the obtained concept description to cover a few negative examples and not covering a few positive examples. The usual solution to the unknown values problem is to suppose that the unknown value is the most frequent value appearing in the examples of the same class to which the current example belongs (Lavrac and Dzeroski, 1994).

Some existing systems using a top-down strategy are the following: CLAUDIEN (De Raedt and Bruynooghe, 1993), MIS (Shapiro, 1983), MOBAL (Kietz and Wrobel, 1992), GRENDEL (Cohen, 1994) and ML-SMART (Bergadano et al., 1988).

Classification of ILP systems

ILP systems can be classified according to four dimensions: incremental/ non-incremental, interactive/non-interactive, single/multiple predicate learning, and theory revision.

- *incremental/non-incremental.* This classification is based on the way in which the examples are obtained. Thus, a system is non-incremental if all the examples are given before learning is performed. A system is incremental if the examples are provided to the system one by one while learning. Non-incremental systems may be top-down or bottom-up whereas incremental systems use a mixture of both strategies. Some incremental systems are MIS (Shapiro, 1983), CLINT (De Raedt, 1992), MOBAL (Kietz and Wrobel, 1992), and CIGOL (Muggleton and Buntine, 1988). Some non-incremental systems are GOLEM (Dolsak and Muggleton, 1992), FOCL (Pazzani and Kibler, 1992), GRENDEL (Cohen, 1994), CLAUDIEN (De Raedt and Bruynooghe, 1993), and LINUS (Lavrac et al., 1991).

- *interactive/non-interactive.* This classification is depending on the external support that a system has. Thus, during the process of learning, interactive ILP systems generate their own examples and ask the user (oracle) about their label (i.e. if the generated examples are positive or negative). They may also ask the user about the validity of the generalisations constructed. Interactivity allows pruning the search space and implies incrementality. Most systems are non-interactive, and the best known interactive systems are CIGOL (Muggleton and Buntine, 1988), MIS (Shapiro, 1983) and CLINT (De Raedt, 1992).

- *single/multiple predicate learning.* In single predicate learning a single predicate is learned from the examples whereas in multiple predicate learning the goal is to learn a set of predicate definitions. Examples of multiple predicate learning systems are MARVIN (Sammut and Banerji, 1986), MIS (Shapiro, 1983), BLIP-MOBAL (Wrobel, 1988), ML-SMART (Bergadano et al., 1988), CIGOL (Muggleton and Buntine, 1988) and CLINT (De Raedt, 1992).

- *theory revision.* The theory revision problem consists of modifying the hypothesis according to new evidence. Theory revision is a usual form of incremental multiple predicate learning. A theory may be inconsistent due to several predicates, for this reason multiple predicate learning is necessary. In Wrobel (1996) an in-depth analysis of the theory revision problem in ILP can be found. Classic theory revisers are the systems MARVIN (Sammut and Banerji, 1986), MIS (Shapiro, 1983), BLIP-MOBAL (Wrobel, 1988), ML-SMART (Bergadano et al., 1988), CIGOL (Muggleton and Buntine, 1988) and CLINT (De Raedt, 1992). The current research tries to build theory revisers without an oracle (as ML-SMART and BLIP-MOBAL).

Existing ILP systems

According to (Muggleton and De Raedt, 1994), there are six systems that have strongly contributed to ILP development: MIS (Shapiro, 1983), MOBAL-BLIP (Kietz and Wrobel, 1992), CIGOL (Muggleton and Buntine, 1988), GOLEM (Dolsak and Muggleton, 1992), FOIL[2] (Quinlan, 1990) and CLAUDIEN (De Raedt and Bruynooghe, 1993). Each one of them has allowed an advance in a special ILP issue and they have been the basis for the construction of other ILP systems.

The Model Inference System (MIS) was the first top-down system that used ILP, introducing techniques as graph refinement, location of incorrect clauses, and manipulation of multiple predicates. Given a language L containing definite clauses and background knowledge BK, MIS uses the following algorithm (from Lavrac and Dzeroski, 1994) to build a hypothesis H:

```
Initialise the hypothesis H to a (possibly empty) set of clauses in L
repeat
   Read the next (positive or negative) example
   repeat
         if there exists a covered negative example e then
              Delete incorrect clauses from H
         if there exists a positive example e not covered by H then
                with breadth-first search of the refinement graph develop a
                clause c which covers e and add it to H
   until H is complete and consistent

   output: hypothesis H
forever
```

MIS interactively accepts new training examples. Clause construction is made searching for a new clause that covers a positive example not covered by the current hypothesis. Search begins by the most general clause and continues by searching clause refinements in a top-down manner obtaining at each step all minimal refinements of the current hypothesis. From the set of minimal refinements those that do not cover the current positive example are rejected. The process finishes when an acceptable consistent hypothesis is found. Some systems based on MIS are CLINT (De Raedt, 1992) and MARKUS (Grobelnik, 1992).

MOBAL (Morik, 1991) is an integrated knowledge acquisition environment consisting of several tools: a model acquisition tool to design rule models that constraints search in the hypothesis space; a sort taxonomy tool that clusters constant terms occurring as arguments in training examples; and a predicate structuring tool that abstracts rule sets to a topology hierarchy. Using these tools MOBAL can constraint the search space.

MARVIN (Sammut and Banerji, 1986) was the first system implementing inverse resolution but without a solid theoretical basis. The CIGOL (Muggleton and Buntine, 1988) system was the first system that formalised the theory of inverse resolution when definite clauses are used. The LOPSTER (Lapointe and Matwin, 1992) system formalises the inversion of the implication.

GOLEM is based on the notion of relative least general generalisation defined by Plotkin. The training examples and the

[2] Muggleton and De Raedt consider FOIL as an ILP system. As we explained before, FOIL's author does not follow this classification, and calls FOIL a relational learning system.

background knowledge are ground facts and function symbols in the terms are allowed. To generate a clause GOLEM randomly takes several pairs of positive examples and their rlgg. The rlgg covering more positive examples is selected and then a new generalisation is attempted. This generalisation is made by taking a new positive example and computing the rlgg of this clause (also a rlgg) and the new positive example. The process finishes when no new positive examples are covered. In post-processing, irrelevant literals are eliminated. To generate more than one clause, GOLEM uses the set covering approach, i.e. builds a clause covering a subset of positive examples, deleting the covered examples from the training set and searching for new clauses that may cover the remaining positive examples.

Since rlgg can contain infinitely many literals or at least grow exponentially with the number of examples, GOLEM uses some restrictions to avoid the growth of rlgg. One of these constraints is the use of determinate clauses[3] in the rlgg body. Negative examples also allow the reduction of the rlgg size. Therefore, if the elimination of a literal does not result in the covering of negative examples, then it is redundant and can be eliminated. This process is repeated until all the redundant literals are eliminated.

The main contribution of FOIL (Quinlan, 1990) is to recognise the expressivity the logic programming as a representation language for inductive learning. This system applies a combination of already known Machine Learning techniques but using a more expressive language. Following the same idea of using classic Machine Learning techniques, LINUS (Lavrac and Dzeroski, 1994) transforms some ILP problems to an attribute representation form, uses a propositional learner and finally translates the results to Horn clauses.

CLAUDIEN (De Raedt and Bruynooghe, 1993) combines data mining principles with ILP. It can be considered the first system that discovers clausal regularities from unclassified data. CLAUDIEN is based on a top-down iterative depth search using refinement under θ-subsumption.

3.2. Lazy Learning

Some authors have defined lazy learning methods as those methods that use extensional descriptions of concepts without generating intensional descriptions. In fact, this is the main difference between inductive and lazy learning methods. Inductive methods produce intensional descriptions of concepts from descriptions of examples (instances).

In lazy learning methods, both the learning process and the process of using the learned knowledge for solving new problems cannot

[3] A clause is *determinate* if each of its literals is determinate. A literal is *determinate* if each of its variables that do not appear in preceding literals has only one possible binding given the bindings of its variables that appear in preceding literals (Lavrac and Dzeroski, 1994).

be considered separately. When solving a new problem P using a lazy learning method, the solution for an old problem (perhaps transformed) is transferred to P. There is an implicit generalisation between the old problem and the new problem P. Thus, lazy learning methods use extensional descriptions of concepts (the instances) and the generalisation step is delayed to the problem solving phase. We will define the lazy learning methods as those that are *problem-centred,* in the sense that the generalisation step is made on-demand, when a new problem has to be solved.

The basis of many lazy learning algorithms is the k-nearest neighbour classifier[4], k-NN, (Dasarathy, 1991) that stores the training set and postpones all effort towards classification decision until problem solving. Given a new example to classify, the k-NN algorithm retrieves the k most similar instances and predicts the class to which the new example can belong. The quality of the k-NN depends on which instances are considered as more similar. The similarity is estimated using the following distance function:

$$d(x,e) = \Big(\sum_{f\in F} \omega(f)\cdot\delta(x_f,e_f)^2\Big)^{1/2}$$

where $e = (e_1, \ldots, e_n)$ is the new example to classify, $x = (x_1, \ldots, x_n)$ is a training example, $\omega(f)$ defines a weighting function and the function $\delta()$ defines how the values of a given feature differ. In particular, $\delta()$ can be defined as follows:

$$\delta(x_f,e_f) = \begin{cases} |x_f - e_f| & \text{f is continuous} \\ 0 & \text{f is discrete and } x_f = e_f \\ 1 & \text{f is discrete and } x_f \neq e_f \end{cases}$$

Notice that d(x,e) is computed for each precedent x, thus precedents can be ordered according their similitude with the new example e. One consequence is that several precedents can be at the same distance of e, therefore a new example can have several solutions.

The result produced by the k-NN is the most plausible class for e, in other words, e is classified as belonging to a class such that $k-NN(e) = \max_{c_j\in J} p(c_j \mid e)$, where

$$p(c_j|e) = \frac{\sum_{X\in K_e} 1(x_c = c_j)\cdot K(d(x,e))}{\sum_{X\in K_e} K(d(x,e))}$$

where $1(x)$ is a function that yields 1 iff the argument x is true and K is a kernel function defined as the inverse of the distance above.

[4] k-Nearest Neighbour algorithm is a noise-tolerant extension of nearest neighbour algorithms (Breiman et al., 1984).

The main shortcoming of k-NN algorithms is its sensitivity to the distance function being used. Nevertheless several variations of the algorithm have been proposed in order to reduce this sensitivity. In (Wettschereck et al., 1997) an in-depth-study of some of these variations and the improvements that they produce can be found.

Lazy learning approaches have been typically associated to analogical reasoning (Carbonell, 1986). Some well known lazy learning methods are derivational analogy, instance-based learning methods and Case-based Reasoning (CBR). Nevertheless, they can also be combined with "pure" inductive methods as decision trees. In particular, LazyDT method (Friedman et al., 1996) builds decision trees in a problem-centred way, constructing the best decision tree for each test instance. In fact, the LazyDT method only needs constructing one path, the one classifying the current instance.

In the next sections two lazy methods such as instance-based and CBR are analysed.

3.2.1. Instance-based Learning

Instance-based learning (IBL) algorithms are derived from the k-NN classifier (Dasarathy, 1991) since they assume that similar instances have similar classifications. IBL algorithms are inspired by the exemplar-based models of categorisation (Smith and Medin, 1981). An exemplar model represents each concept as a set of exemplars, where each exemplar may be either a concept abstraction or an individual instance of the concept. The main characteristics of exemplar models are: 1) concepts are not represented as a set of necessary and sufficient conditions abstracted from exemplars, 2) descriptions are disjuncts, and 3) properties of a concept are a function or a combination of the properties of exemplars. Smith and Medin proposed two basic exemplar models, the *proximity model* and the *best examples model.* The *proximity model* stores all the training instances without performing any abstraction over them. Thus, a new instance is classified by computing its similarity with each one of the existing instances. The new instance is classified as belonging to the concept of the more similar instance. A variation of this model is the *best examples model* in which only the most typical instances (prototypes) of each concept are stored.

One of the early works on IBL was made by Kibler and Aha (1987). At that time, there was much research about how to generalise concepts from examples but there was not much research about directly using the stored examples. The primary output of IBL algorithms is a concept description, in the form of a function that maps instances to concepts. An instance-based concept description includes a set of stored instances and, possibly, some information concerning their past performance during classification (e.g., their number of correct and incorrect classification predictions). This set of instances can change after each training set instance is processed. Concept descriptions are determined by how the similarity and classification functions use the training instances. The following algorithm, called IB1, is the simplest IBL algorithm (from Aha et al., 1991):

```
concept-description := ∅
for each x ∈ concept-description do
      for each y ∈ concept-description do
            Sim[y] := Similarity (x, y)
      end-for
      y_max:= some y ∈ concept-description with maximal Sim[y]
      if class (x) = class (y_max)
           then classification := correct
           else classification := incorrect
      end if
      concept-description := concept-description ∪ {x}
end-for
```

Where Similarity $(x,y)=\sqrt{\sum_{i=1}^{n} f(x_i,y_i)}$ is the similarity function assuming that instances are described by n attributes. The function f is defined as follows:

$$f(x_i,y_i)=\begin{cases}(x_i-y_i)^2 & \text{for numeric - valued attributes}\\ 1 & \text{if } x_i \neq y_i \text{ for boolean and symbolic - valued attributes}\\ 0 & \text{if } x_i = y_i \text{ for boolean and symbolic - valued attributes}\\ 1 & \text{if both } x_i \text{ and } y_i \text{ are missing attributes}\end{cases}$$

IB1 is identical to the nearest neighbour algorithm except that it normalises the ranges of attributes, processes instances incrementally, and has a simple policy for tolerating missing values.

The model presented by Kibler and Aha (1987) does not store all the instances and also avoids the prototype construction process. Their proposal was the *Grow (additive) algorithm* and the *Shrink (subtractive) algorithm*. The growth algorithm (also called IB2) is like the IB1 algorithm but only stores instances that have not been correctly classified. The Grow algorithm reduces storage space but its main shortcoming is that the classification accuracy decreases, especially when domains are noisy or have many exceptional instances.

The Shrink algorithm (also called IB3) tries to classify each instance from the others. If it does, the instance is not to be stored. The idea behind this algorithm is that the system accuracy is not necessarily better if all the instances are stored. In fact, the system accuracy may decrease if the stored instances are not typical. The Shrink algorithm is highly sensitive to the number of irrelevant attributes used to describe the instances. In (Aha et al., 1991) an in depth analysis of the behaviour of the three algorithms (IB1, Grow and Shrink) can be found.

The main advantages of IBL algorithms are its simplicity, its support to relatively robust learning algorithms (allowing noise and irrelevant attributes), and its relatively low updating costs.

Emde and Wettschereck (1996) have proposed the adaptation of instance-based methods to ILP representation. They have developed the Relational Instance-Based Learner (RIBL) algorithm that has been implemented as an external tool of MOBAL. RIBL first generates cases from the examples. Each case (represented as a conjunction of literals) contains an object, its description and its relations with other objects. The similarity between cases is estimated using a modified version of Bisson's algorithm (1992). Finally, a generalised form of k-NN algorithm that handles relational representation is applied.

Instance-based learning methods, like inductive methods, can improve their behaviour by means of an accurate selection of attributes describing the instances. Instance-based learning methods can also be improved by introducing constructive induction in order to adequate the instance language and the language required to represent concept descriptions. An instance-based method introducing constructive induction of features is IB3-CI (Aha, 1991).

3.2.2. Case-based Reasoning

The experience in solving problems is a human quality that is necessary to capture in order to create a model of intelligent behaviour. Research in *Case-based Reasoning* (Kolodner, 1983) began at the 80's as an attempt to provide a problem solving model that was closer to psychological models than ruled-based systems. The cognitive model behind Case-based Reasoning (CBR) was inspired by *scripts* (Schank and Abelson, 1977) and by *dynamic memory* models of cognition (Schank, 1982).

CBR is associated to problem solving experiences and it is based on the human capability to solve new situations according to the similarity among the new situation and past situations already solved. Thus, when a new situation is similar to one or several old situations, the decisions taken and the knowledge contained in old situations provide a starting point to interpret or solve the new situation. In this way a new situation is solved taking advantage of inferences and decisions already made in past situations. Usually, each experience is considered a *case*. Experienced situations are *past cases* which may be reused to solve new problems and a *new case* is the description of a new problem to be solved.

CBR assumes that cases, give operational knowledge, i.e. instances show how the knowledge can be used and applied. General knowledge has advantages such as the ability to use a domain model capable to deal with most of situations. Nevertheless, sometimes general knowledge is too abstract to be applied to a particular situation and sometimes not all the needed knowledge is available. Another shortcoming of general knowledge is the difficulty to deal with exceptional situations. CBR can handle specific knowledge that is difficult to obtain using a general domain model and, in addition, general domain knowledge (rules or domain models) can be also used. On the other hand, systems using only general knowledge do not learn from experience, whereas in CBR each new solved case may be used to solve future problems.

CBR represents an integrated approach for learning and reasoning. A case-based reasoner learns from each problem solving session by storing the relevant information about the solved problem, and converting it in an available experience to solve future problems. Learning becomes a process of extracting relevant information from past problem solving experiences. Each new problem (case) has to be indexed in the system's knowledge structure in a way that can be easily retrieved when a similar problem is encountered later. Thus, from a new case a case-based reasoner can learn several thinks: the solution, the adaptation method used, relations with other cases, possible failures, etc.

The quality of CBR depends on several issues: 1) the experiences (cases) available, 2) the ability to understand new situations in terms of past situations, 3) the ability to adapt old solutions, 4) the ability to evaluate and fix old solutions, and 5) the capability to integrate new experiences.

CBR offers some advantages with regard to rule-based reasoning. The first one is that a library of cases seems more similar to the human expertise than a set of rules. A second advantage is that in real-world problems, cases and their solutions are easily acquired whereas it may be extremely difficult to specify all the necessary rules. On the other hand, CBR can be considered as a methodology complementary to model-based reasoning (MBR). MBR is used in domains where the causality can be well represented whereas CBR does not need a complete domain knowledge to produce correct results. Another difference is that MBR uses general knowledge whereas CBR uses specific knowledge. There are several systems, such as CASEY (Koton, 1988) and KRITIK (Goel, 1991; Goel and Chandrasekaran, 1989) integrating MBR and CBR.

The main advantages of CBR with respect to rule-based and model-based reasoning are the following: 1) it is applicable in domains with incomplete knowledge; 2) it obtains solutions efficiently; 3) it may solve problems when not algorithmic methods are available; 4) it avoids possible problems that have already appeared; and 5) it focuses on the most important parts of a new case.

A main disadvantage of CBR is that generally it does not completely explore the whole space of solutions. As a consequence, locally optimal solutions may be found that are not the global optimal solution. Other disadvantages of the CBR are related to the bias introduced by the cases in the library, the set of retrieved cases and the blind use of old cases (i.e. the validity of an old case for the new case is not evaluated).

Case-based Reasoning is useful in a wide variety of problem solving tasks, such as planning, design and diagnosis. Planning is the process of obtaining a sequence of steps (i.e. a plan) or a schedule to achieve some goals. A planner has to assure that preconditions of each step are preserved before performing a new step. CBR handles these problems reusing old plans (Alterman, 1986; Hammond, 1989; Goodman, 1989; Haigh and

Veloso, 1995; Aarts and Rousu, 1996; Muñoz-Avila and Hüllen, 1996). The proposed plan has to be adapted to the new goals.

Design problems are commonly defined as a set of constraints over a collection of possible components. The solution of design problems is a composite object satisfying all the problem constraints. Problem constraints can underspecify the problem and thus several solutions are possible. Sometimes problem constraints overspecify the problem and no solution is possible without relaxing some constraints. CBR suggests new designs from old designs created under similar constraints. These past cases can be adapted to obtain a new design satisfying all the desired constraints (Sycara, 1988; Navichandra, 1988; Hinrichs and Kolodner, 1991; Smyth and Keane, 1995; Bartsch-Spörl, 1995; Hurtley, 1995; Surma and Braunschweig, 1996; Zdrahal and Motta, 1996).

Diagnosis is a particular explanation problem where a problem solver has to explain a set of symptoms. A case-based diagnostician can use cases to suggest explanations for symptoms and to prevent inappropriate explanations (Leake, 1992; Bareiss et al., 1989; Netten and Vingerhoeds, 1995; Portinale and Torasso, 1995; Lenz et al., 1996).

In the next sections we explain the main CBR concepts and main CBR tasks following the CBR cycle defined in (Aamodt and Plaza, 1994): *retrieval*, that obtains precedents similar to the current problem and chooses the best precedent; *reuse*, that decides if the solution of the best precedent may be applied to the current problem or how it has to be adapted; *revision* that assesses the goodness of the adapted solution; and *retain* that decides which parts of the new case may be useful for solving new problems.

Library of Cases

Kolodner (1993) provides the following definition for cases:

> "A case is a contextualised piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner"

A case contains the description of a problem according to the goals to be achieved, restrictions about these goals, characteristics of the problem and relations between parts of the problem. Cases have been represented using different notations. Thus, CASEY (Koton, 1988) and PROTOS (Bareiss, 1989) use an attribute-value representation. MEDIATOR (Simpson, 1985) uses structured representations as frames. CHEF (Hammond, 1989) uses a hybrid representation: cases are organised as frames but slot fillers of the frame are represented using first-order predicate calculus.

The *library* (or *memory*) *of cases* contains the expertise of a case-based reasoner. The structure, content and organisation of solved cases are essential for reuse. This organisation has to be dynamic since the incorporation of a new case can influence the organisation of already existing cases. There are two possible organisations of the library of cases: flat and hierarchical. In *flat* organisation, cases are sequentially stored using a list or a file. In *hierarchic* organisation cases are organised according to their characteristics. Each node of the hierarchy corresponds to a characteristic shared by a set of cases. This organisation is appropriate when the library has a big size.

Cases are organised in the library using *indexes*. The determination of useful indexes is one of the main issues of the CBR. Some properties that indexes have to satisfy are predictivity, utility, abstraction and concretion. Thus, indexation has to allow reuse of appropriate cases supporting the achievement of the task goals (predictivity and utility). On the other hand, indexes have to be both abstract enough to make a case useful in a variety of future situations, and concrete enough to be easily recognisable in future situations. Using indexes, a case-base reasoner can retrieve a subset of cases that are potentially useful to solve the new situation.

Retrieval Task

The main problem of a case-based reasoner is to determine past situations relevant (similar) to solve a new situation. Past situations have to be labelled and organised in the library of cases in order to be retrieved using the features of the new situation. The retrieval task has as input a (partial) problem description and provides a past case or a ranking of past cases having the best matching with the new problem. An important issue of the retrieval task is to determine useful indexes to retrieve past cases. Indexes may be the input features of the new case. Nevertheless, for knowledge-intensive methods (using general domain knowledge) more elaborated indexes can be determined.

The organisation of the library influences the retrieval of old cases. In flat organisation cases are retrieved using a matching function that is sequentially applied to each case. The main advantage of this scheme is that all the library is explored and the retrieval quality depends on the matching function. The main disadvantage is that the retrieval time increases drastically when the number of cases clearly increase. When the library of cases has a large size, the hierarchic organisation is recommended. The main advantage of this scheme is the efficiency in the retrieval process. The main disadvantages are the difficulty in adding new cases and the memory space that is used.

The similarity between cases may be evaluated in two ways: syntactically and semantically. The *syntactic similarity* uses as indexes the problem descriptors. This approach is appropriate for domains having not much general knowledge available. An example of system using syntactic similarity is CYRUS (Kolodner, 1983). The *semantic similarity* (also called *knowledge intensive*) uses more elaborated indexes obtained by applying general domain knowledge to case descriptors. This approach is used in PROTOS (Bareiss, 1989), CASEY (Koton, 1988) and CREEK (Aamodt, 1991).

Similarity assessment may be knowledge-intensive, using the process of understanding the new case as guide for the matching. Another option is to weight the problem features according to their importance for characterising the problem during the learning phase. The similarity assessment task can obtain a set of cases as best match. In this situation, an explanation justifying non-identical features for each past case of this set may be generated. When a strong enough explanation is found, the corresponding past case is selected as the best match.

Reuse Task

The reuse of a past case solution in the context of the new case focuses on two aspects: the differences among the past cases and the new case, and which part of the retrieved case can be transferred to the new case. In classification systems, the solution of the retrieved case is directly the solution of the new case. However, other systems need a transformation process (adaptation) of the solution. Adaptation processes modify an old solution to provide a new one. The kind of adaptation varies according the differences between the new case and the retrieved one.

There are two kinds of reuse (Aamodt and Plaza, 1994): transformational and derivational. In *transformational reuse* the past case solution is not directly a solution for the new case but there exists some knowledge in the form of transformational operators such that applied to the old solution they transform it into a solution for the new case. Transformational reuse focuses on the equivalence of solutions, and this requires a strong domain-dependent model containing transformational operators and how they can be applied.

Another approach is when the retrieved case holds information about the method used for solving it (including a justification of the operators used, subgoals considered, alternatives generated, failed search paths, etc). From this information *derivational reuse* reinstantiates the retrieved method to the new case and executes it into the new context.

Revision Task

During the revision task the solution generated by the reuse task is evaluated. This evaluation is made by asking a teacher or by a real world simulation (as the CHEF system (Hammond, 1989)). If the evaluation result determines that the proposed solution is correct, the new case may be retained. Otherwise, an opportunity to learn from failure appears since the solution case has to be repaired. Case repair involves detecting the errors of the current solution and retrieving or generating explanations for them. Errors are repaired using general causal knowledge and domain knowledge about how to avoid or compensate errors in the domain. If the revision task assures the correctness of the repaired solution, this can be retained. Otherwise the repaired plan has to be in turn evaluated and repaired (Hammond, 1989).

Retain Task

The retain task determines which information about the new case can be useful for solving new cases and incorporates it to the existing knowledge. Relevant problem descriptors and problem solutions are useful information but explanations of why a solution is successful or not may also be useful to store.

A main issue in retaining new cases is the *indexing problem*, i.e. to decide what type of indexes to use for future retrieval and how to structure the search space of indexes. The approach used by syntax-based methods is using all input features as indices. Other approaches, as the used in CASEY (Koton, 1988) also use as indices the observed features.

The integration of a new case in the library of cases implies a modification of the existing indices. The first consideration about a successful case is to consider if its incorporation to the library of cases may be useful. Notice that the systematic incorporation of all the new cases can increase the system's inefficiency. Commonly, the future utility of the successful case is estimated before deciding whether it has to be incorporated in the library. If a successful case is similar to an existing case the construction of a prototype or scheme generalising both cases can be considered. From a failed case, the system can learn from failure by modifying the corresponding indexes in order to prevent the retrieval of the same case in a similar situation.

Usually, failed cases produce the adjustment of the index strengths for a particular case or solution.

4. Knowledge Modelling, Learning and Problem Solving Integration

Some authors consider that KBS development can be divided in two phases: Knowledge Aquisition phase and Problem Solving phase. During the Knowledge Acquisition phase all the necessary knowledge to solve problems in a domain is acquired. As we have seen (section 2), this knowledge may be acquired using Knowledge Modelling methodologies. During the Problem Solving phase the acquired knowledge is used to solve new problems. Nevertheless, to accurately solve problems in a domain is also necessary a maintenance of the acquired knowledge (Aamodt, 1991). In other words, the problem solver could acquire new knowledge from the experience in solving new problems. This new knowledge is usually acquired using learning methods.

In fact, there are different kinds of knowledge (general domain knowledge, cases, strategic knowledge, etc) that can be used to solve problems in a domain and each kind of knowledge can be acquired in a different way. For instance, cases can easily be acquired from a domain expert using a knowledge acquisition methodology. Instead general domain knowledge or strategic knowledge may be more difficult to acquire from the experts during the Knowledge Acquisition phase. So, this kind of knowledge can be easily acquired using a learning method that learns from experience.

Currently, some systems integrating Knowledge Acquisition, Learning and Problem Solving have been developed. These integrated systems are usually based on a system architecture that allows the explicit expression of various types of knowledge relevant to a particular application. Such architectures also contain problem solving and learning components which are able to effectively utilise a continually improving body of knowledge. In the next sections two of such architectures are analysed. The first one, CREEK, has a reasoning process based on a combination of three reasoning types: model-based reasoning, CBR and rule-based reasoning. The second architecture, MUSKRAT, integrates different knowledge acquisition methods (such as learning, knowledge elicitation and knowledge base refinement tools), with several problem solving methods.

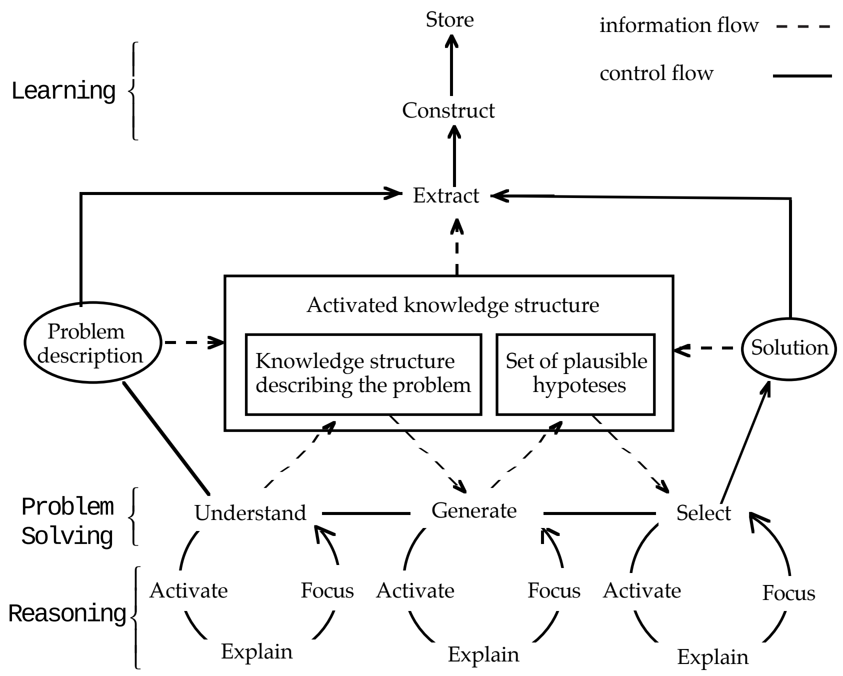


Figure 2.7. The CREEK model of integrated learning, problem solving and reasoning (from Aamodt, 1991).

4.1. CREEK

CREEK (Aamodt, 1991) is a knowledge intensive approach to problem solving and learning, based on an intensive use of the domain knowledge in both problem solving and learning methods. The basis for the structure and functionality of CREEK is the framework shown in figure 2.7. The framework specifies a set of general requirements concerning knowledge modelling, problem solving and sustained learning[5].

The problem solving model of CREEK has three phases: 1) *Understanding* the problem, 2) *Generating* plausible solutions, and 3) *Selecting* a good solution. Problem understanding consists of interpreting the input description in terms of the conceptual model. Then the system tries to integrate this input into the existing knowledge. At the second phase, the system obtains a set of possible solutions that have been justified (in the sense that they achieve the goal without contradicting important constraints). The last phase is an evaluation of plausible solutions in order to select a good solution to the problem.

Reasoning in CREEK is viewed as a subprocess of problem solving. Given some findings and a goal, a reasoning process may be described by three sub-processes: 1) *Activating* knowledge structures, 2) *Explaining* candidate facts, 3) *Focusing* on a conclusion. Thus, the reasoning process is viewed as a process of activating a certain part of the existing knowledge (including triggering of hypotheses and goals), explaining to what extent activated parts form a coherent knowledge structure, and focusing within the explained structure in order to finally return an explicit answer. This reasoning model can also describe multi-paradigm reasoning since the three phases model may be applied to each reasoning method separately.

The learning model integrates a case-based learning approach and an explanation-based learning method with a learning apprentice approach. Thus, learning and problem solving are thighly integrated. The learning process may be described by the following steps: 1) *Extracting* learning sources, 2) *Constructing* new knowledge structures, 3) *Storing and indexing*. The extracting learning sources is active throughout problem solving, and its task is to keep track of information and knowledge that will later be used as sources learning. The step of constructing new knowledge structures is based on methods for constructing cases, for constructing general rules from a set of cases (using EBL), and for modifying knowledge structures by integrating new knowledge that has been inferred or accepted from the external environment. The storing and indexing steps make available the new knowledge for future problems.

[5] Aamodt defines *sustained learning* as the ability to learn from problem solving experience, continually improving the knowledge by updating the knowledge base after each problem solving session.

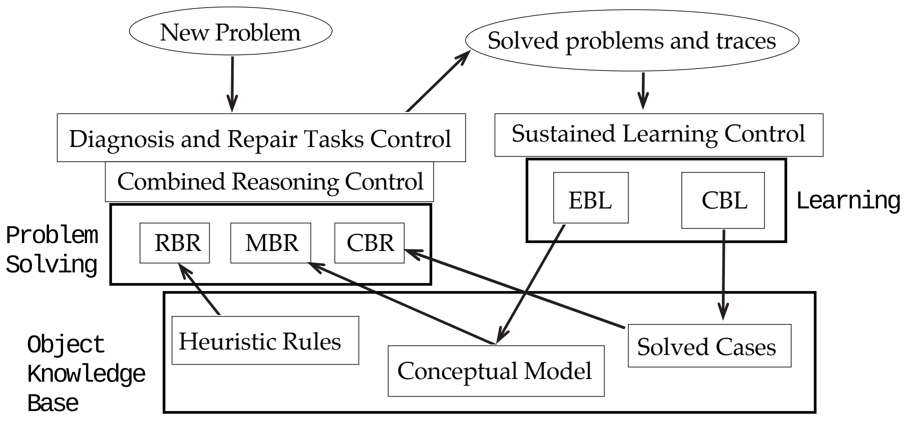


Figure 2.8. Functional architecture of CREEK (from Aamodt, 1991). RBR stands for Rule-based Reasoning; MBR stands for Model-based Reasoning; CBR stands for Case-based Reasoning; EBL stands for Explanation-based Learning; and CBL stands for Case-based Learning.

Based on the above model, the functional architecture of CREEK (figure 2.8) contains three building blocks: an object level KB, a problem solver, and a learner. The KB contains a conceptual knowledge model, a collection of past cases, and a set of heuristic rules. Problem solving can use a combination of model-based, case-based and rule-based reasoning. Finally, the learner combines case-based learning and explanation-based learning methods to improve the problem solving behaviour.

4.2. MUSKRAT

MUSKRAT (Sleeman and White, 1996) is a open architecture which supports the integration of problem solvers and knowledge acquisition tools (knowledge elicitation, machine learning and KB refinement tools), and assists the user to select the most suitable knowledge acquisition tool. MUSKRAT takes the idea of user's assistance from the Consultant tool. Consultant (Craw et al., 1992) is an advice-giver system that asks the user about the task he wants to solve, the data and background knowledge he can provide, etc, and recommends one or more suitable learning tools. The main shortcoming of Consultant is that the user must first decide what knowledge is required to solve his problem and then Consultant can help him with the choice of a suitable tool. In other words, Consultant is told what the user wants, not what he wants to do. MUSKRAT is specially focused on knowledge acquisition, supporting the user in the selection of an appropriate tool to acquire the necessary knowledge. The selection of one or several KA techniques proceeds as follows:

1. Identify an application task, i.e. a problem to be solved in a particular domain
2. Select a suitable problem solver to solve this task. If no single problem solver is found the task is decomposed in subtasks where each subtask can be solved by a specific problem solver (PS)
3. Determine the required KB for each problem solver
4. For each KB compare requirements of the problem solvers with available knowledge sources and define one or more KA tasks
5. Select the appropriate KA tools to solve the KA tasks
6. Apply the selected KA tool.

MUSKRAT supports steps from 3 to 6. The architecture (see figure 2.9) is organised around a set of KB's, that is the interface between the KA tools and the problem solvers. A KB is defined as a body of knowledge required by a problem solver.

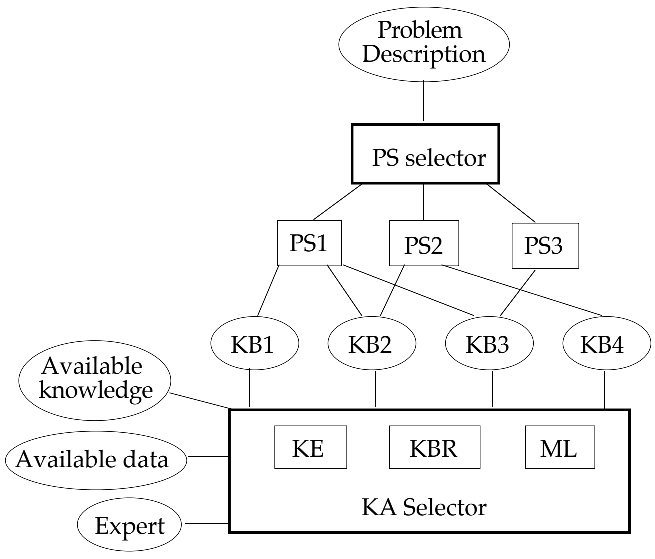


Figure 2.9. The MUSKRAT Architecture. PS stands for Problem Solving; KBR stands for Knowledge Base Refinement tool; KE stands for Knowledge Elicitation tool and ML stands for Machine Learning tool.

This architecture can be used at two different stages of the problem solving cycle: 1) the selection stage of suitable tools, and 2) the use of tools to acquire knowledge and solve the problem. Tool selection process uses the PS selector to choose a suitable problem solver. Once a problem solver has been chosen MUSKRAT knows which KB are required. The next step is to identify available knowledge resources which can belong to three categories: available knowledge sources, available data and human experts. Available knowledge sources refers to knowledge that is already in the form required for a KB (for example as a set of rules). Available data refers to data relevant to the problem and from which useful information could be extracted, although it is not in the form required by the KB. Typically this may consist of past cases or useful system models to perform diagnosis. An expert is a person who can provide various forms of knowledge, possibly with the help of a KE tool and/or a knowledge engineering.

The KA selector is the central component of MUSKRAT. It compares the requirements of the selected problem solver with the characteristics of available knowledge sources and recommends the use of one or more KA tools. For that purpose it has a knowledge level description of each available KA tool and performs a means-ends analysis to decide which one is most capable of reducing differences.

In MUSKRAT all the KB are expressed in CKRL, an information interchange language (Morik et al., 1991). This language, that is not directly executable, consists of declarations that can be translated into the internal representation of different tools. The use of a uniform knowledge representation facilitates knowledge sharing and reuse since a KB can be used by several problem solvers. It also allows the integration of new problem solvers and KA tools into MUSKRAT at the cost of implementing a single interface to or from CKRL.

When no KA tool is available to produce the required KB, the KA selector describes the requirements to an expert and assumes that he will be able to provide this knowledge. MUSKRAT does not support the use of individual tools: its role is limited to the communication of knowledge between different tools. The user has the responsibility to evaluate the solution obtained by the problem solver and, if necessary, to start a new KA cycle.

5. Multistrategy Learning Systems

Machine Learning (ML) research has shown that there is no single method that could be considered as the best for all domains. Developed ML algorithms use different representation formalisms and search algorithms that provide different results for different application domains. Thus, empirical induction requires many input examples and small amount of background knowledge; EBL requires one input example and a complete background knowledge; learning by analogy and case-based learning require background knowledge allowing new inferences about case properties; and learning by abduction requires causal background knowledge related to the input. This suggests that real-world learning problems could be solved by systems that can apply different strategies in an integrated way.

Multistrategy learning systems (Michalski and Tecuci, 1991) are learning systems containing several learning methods. Some multistrategy learning systems combine several learning methods in order to obtain the best descriptions. Examples of these systems are: UNIMEM (Lebowitz, 1986), that uses EBL to focus similarity-based learning; AQ17-MCI (Wnek and Michalski, 1991), based on the *Inferential Theory of Learning* proposed by Michalski (1991), that combines two strategies of inferential learning (empirical induction and deduction) and two computational methods (data-driven and hypothesis-driven); and EITHER (Mooney and Ourston, 1991), that uses analytical methods (deduction and abduction) to identify incorrect parts of the theory and empirical methods (induction) to determine the corrections required by the theory.

Other multistrategy learning systems integrate several independent methods, selecting the most appropriate of them according to the learning goal. An example is the Meta-AQUA architecture (Cox and Ram, 1994) that takes the learning problem as a planning problem. Each planning goal is a learning goal achieved using a method capable of producing changes in the background knowledge. Learning goals specify the structure, knowledge contents, and organisation of the knowledge in memory. Some learning goals considered by Meta-AQUA are knowledge refinement goals, knowledge expansion goals, knowledge differentiation goals, knowledge reconciliation goals, and knowledge organisation goals.

There are multistrategy learning systems that are toolkits of techniques where the user can choose the appropriate methods according to the application domain. An example is LINUS (Lavrac and Dzeroski, 1994) that integrates various attribute-value inductive learners (a decision tree induction system and two rule induction systems) within a common ILP framework which allows relational learning in presence of background knowledge. Finally, a different technique is meta-learning (Chan and Stolfo, 1993), that uses several techniques (including parallelism) and combines the results obtained from each one of them. The advantage of the meta-learning is that is able to manage large sets of data.

The above mentioned multistrategy learning systems have a predefined order in which the methods integrating the system are applied. A more dynamic approach is the Multistrategy Task-adaptive Learning proposed by Tecuci (1991). The goal of this system is to build a justification tree proving that the input is a plausible consequence from the KB. Each level of the justification tree can be constructed using different types of inference: deduction, analogy, abduction or inductive generalisation. Therefore, the generalisation of the justification tree provides a different strategy according the current problem.

In the following sections we explain in more detail EITHER, LINUS and MTL.

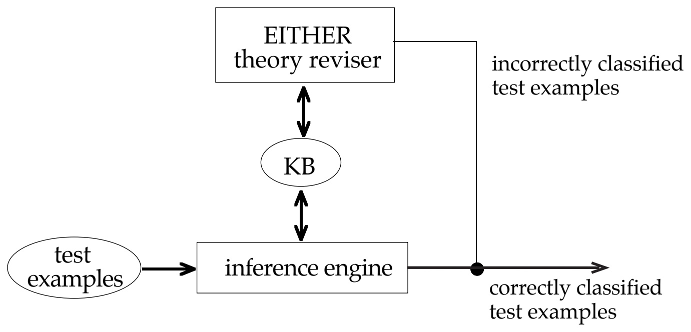


Figure 2.10. The EITHER architecture (from Mooney and Ourston, 1991).

5.1. EITHER

EITHER (Mooney and Ourston, 1991) is a multistrategy learning system that uses independent modules for deductive, abductive and inductive reasoning to revise an incorrect domain theory. EITHER can be viewed as a system integrating analytical methods (deduction and abduction) and empirical methods (induction). The analytical part of the system is used to identify failing parts of the theory and to constraint the examples used for induction. The empirical part determines specific corrections of failing rules in order to make them consistent with the training examples. Figure 2.10 shows the EITHER architecture. The purpose of EITHER is the following:

Given an imperfect domain theory from a set of concepts (categories) and a set of classified examples each described by a set of observable features

Find a minimally revised version of the domain theory that correctly classifies all the examples.

The formalism used by EITHER is Horn clause logic with the closed world assumption. In EITHER Horn theories can be defined by a directed acyclic graph. The criterion used by EITHER to assure a minimal revision of the theory is based on syntactic measures such as the total number of symbols added or deleted.

The deductive component of EITHER is a standard backward chaining, Horn-clause theorem prover similar to Prolog. Deduction is the first step in theory revision. The system tries to prove whether each example belongs to some known concept. Failing positives (examples that the system cannot prove that they belong to the correct concept) indicate overly-specific aspects of the theory and are passed on the abductive component. Failing negatives (examples proved as belonging to an incorrect concept) indicate overly-general aspects of the theory and are passed on to the specialisation procedure. The deductive component is also used to classify unseen examples.

The abductive component is used to solve the problem of the failing positives, i.e. those positive examples that have not been proved due to the specificity of the theory. The abductive component uses exhaustive search to find all the partial proofs of each failing positive example. This search is useful to detect which antecedents prevent the example to be proved. In a complex problem, there are many partial proofs for each failing positive. EITHER tries to find the minimum number of antecedents to be retracted to fix all the failing positives, in order to assure a minimal change in the theory.

The inductive component of EITHER is used to learn new rules or new antecedents when retracting an antecedent element causes new failing negatives or when the retraction of a rule element causes new failing positives. If the antecedent retraction produces an over-generalisation, the inductive component is used to learn a new set of rules for the corresponding concept. If the rule retraction produces an over-specialisation, the inductive component is used to learn additional antecedents to add to the rule instead of retracting it. EITHER uses the ID3 algorithm as inductive component and later translates the decision tree into a set of rules. Inverse resolution operators (Muggleton, 1987; Muggleton and Buntine, 1988) are also used to introduce new intermediate concepts and produce a multi-layer theory from a translated decision tree.

In (Mooney and Ourston, 1991) the results of EITHER over two real expert-provided rule bases, one in molecular biology and another in the Soybean domain, can be found.

5.2. LINUS

LINUS (Lavrac and Dzeroski, 1994) is an ILP toolkit of learning algorithms (see figure 2.11) that induces hypotheses in form of constrained deductive hierarchical database (DHDB) clauses. LINUS transforms the task of relational learning for finite domains into a propositional learning task. Propositional learners included in LINUS are ASSISTANT (Cestnik et al., 1987), NEWGEM (see in Lavrac and Dzeroski, 1994)) and CN2 (Clark and Niblett, 1989). An interface transforms training examples from the DHDB form to attribute-value tuples form in such a way that the propositional learners in LINUS can be applied. The result of propositional learners are if-then rules that will be transformed back into DHDB clauses. The translation of the relational learning into propositional learning takes advantage of some advances made in propositional learning, for instance in handling imperfect data. The LINUS learning algorithm has the following steps:

1. Pre-process the training set to establish the sets E^+ and E^- of positive and negative examples

2. Use of background knowledge to transform examples in E^+ and E^- from the DHDB form to attribute-value tuples
3. Induce if-then rules from the tuples using an attribute-value learner
4. Transform the obtained if-then rules into DHDB clauses
5. Post-process the obtained clauses to generate a hypothesis

Training examples in LINUS are ground facts. Background knowledge is composed of deductive database clauses (possibly recursive). The hypothesis language is restricted to constrained DHDB clauses with typed variables and without recursive predicates. The pre-process includes the generation of negative examples and the handling of missing values. In noisy and inexact domains the negative examples have to be explicitly given.

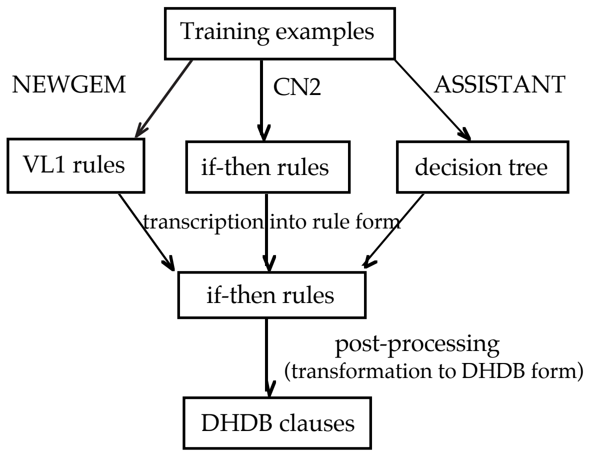


Figure 2.11. The LINUS architecture (from Lavrac and Dzeroski, 1994).

DINUS is the algorithm translating the relational form to the propositional form. This algorithm constructs two lists, one containing determinate literals that introduce new variables (namely L), and the other (F) containing all the literals using predicates from the background knowledge. The difference list F-L contains the list of attributes that can be used for propositional learning (see details and algorithm in Lavrac and Dzeroski, 1994).

Once DHDB clauses have been translated into attribute-value tuples, the propositional learner chosen by the user is executed. Later a post-process eliminating irrelevant literals and translating the obtained hypotheses to DHDB clauses is applied. The elimination of irrelevant literals from clauses constituting a hypothesis makes the induced hypothesis more compact and more accurate when classifying new cases. In noisy-free domains, a literal L in a clause c is irrelevant if the clause c', obtained by eliminating L from c, does not cover negative examples. In noisy domains, L is irrelevant if it can be removed from the body of the clause without decreasing the classification accuracy of the clause on the training set.

In order to compare the LINUS behaviour with FOIL, it was tested in four relational domains, such as learning family relationships (Hinton, 1989), learning the arch concept (Winston, 1975), the Eleusis game (Dietterich and Michalski, 1986) and learning illegal chess endgame positions (Muggleton et al., 1989). Otherwise, LINUS has also been applied to real-world domains such as diagnosis of rheumatic diseases (in (Lavrac and Dzeroski, 1994) a brief description of this domain can be found), mesh design (Dolsak and Muggleton, 1992) and learning qualitative models of dynamic systems, such as the U-tube. In (Lavrac and Dzeroski, 1994) a detailed comparison of LINUS results against results produced by other ILP systems (i.e. GOLEM, FOIL, mFOIL) over the same domains can be found.

5.3. Mutistrategy Task-adaptive Learning

The main idea of the Multistategy Task-adaptive Learning (MTL) approach (Tecuci, 1991) is to define an architecture for a learning system at a level of abstraction that would allow a common view on the single-strategy learning methods, and would therefore facilitate their dynamic integration. The goal of MTL is to identify basic inference mechanisms (deduction, abduction, analogy, determination, etc) and define a new multistrategy system combining all those reasoning and learning capabilities. This new method (considered as a combination or integration of methods) has to dynamically integrate all the elemental reasoning mechanisms according to the new problem to solve.

The MTL system has two main steps: understanding the input and generalising that understanding. Input understanding consists of building a plausible justification tree proving that the input is a plausible consequence of the KB. First, the system tries to justify a given predicate by deduction. If this attempt succeeds, then the justification of the given predicate is reduced to the justification of other predicates. However, if this attempt fails, the system tries to justify the given predicate by using as many plausible reasoning methods as possible. Methods are tried according to the following order: justified analogy, abduction and inductive generalisation. If one of them produces a plausible inference step, then the system tries to use the remaining ones in order to confirm or contradict it. If no contradiction is found, the inference step is accepted. All the knowledge related to the tree construction is added to the KB. When the system cannot build a justification tree proving the input, this input is stored in the KB.

The generalisation of the justification tree is made by analysing individual inference steps and determining if they could be locally generalised within the constraints of the KB used to make these steps. After a local generalisation of the inference steps, the system unifies them globally, and builds a generalised justification tree. The idea is to replace each inference step S with the generalisation of all similar inference steps that could be derived from the knowledge produced by S. Thus, deductive steps are replaced by the deductive rules generated by them; analogy steps are generalised by considering the knowledge used to derive them; and the generalisation of inductive steps depends on the type of induction performed. In general, when an inference step is a result of different types of inference, all the involved knowledge is used to generalise this inference step.

When MTL learns from several examples the justification tree has to be generalised and/or specialised in order to make it consistent with positive and negative examples. A negative example explained by a justification tree means that the tree has to be specialised since it contains some incorrect inferences. The specific incorrect inferences are detected in MTL using the credit assignment problem with some restrictions. For example, by assuming that 1) only one step is incorrect, or 2) the inference is incorrect due to an incorrect left hand rule. MTL also has several criteria to select which is the inference step that should be modified when more than one are detected as incorrect. The order of these selection criteria is the following: 1) abduction, analogy and deduction; 2) selection of those inference steps producing the minimal coverage changes in previous examples; 3) selection of the inference steps producing a minimum increase of complexity; and 4) arbitrary selection of one of the remaining inference steps.

One of the major advantages of the MTL is that it enables the system to learn in situations in which single-strategy learning methods, or some of the previous combinations of methods that have been found were insufficient. Therefore, the proposed approach has a great potential to make machine learning programs applicable to a wider range of problems. Another important aspect of the method is that it behaves as a single-strategy method, whenever the applicability conditions for such a method are satisfied. In this respect, the proposed MTL method may be regarded as a generalisation of the single-strategy methods.

6. Context of our Work

Solving problems in complex and real-world domains needs several kinds of knowledge and several methods to acquire this knowledge. Moreover, the acquired knowledge has to be updated according to the experience in order to accurately solve new problems. Our goal is to define a framework supporting the integration of Problem Solving and Machine Learning. To achieve this goal we need a representation in which both problem solving and learning methods may be integrated. Our proposal is to use Knowledge Modelling methodologies as a tool for such integration.

Knowledge Modelling methodologies allow the analysis of problem solving methods (PSM) in a domain. Each PSM is represented by a series of tasks, models and methods (some Knowledge Modelling methodologies have several methods associated to solve a specific kind of task, but when a method is assigned to a task it cannot be changed). In our framework, a task can be solved by means of several alternative methods. During problem solving the appropriate method is selected in a lazy problem-centred way. In other words, each method has some applicability conditions that are evaluated during problem solving taking into account the new problem to solve. Thus, for each task (or subtask) a different method can be selected in a problem-centred way.

The goal of a task may be the acquisition of new knowledge. This kind of task may be solved either by knowledge elicitation from an expert or by means of a learning method. Some integrated systems have several Knowledge Acquisition tools and are capable of detecting which of these tools is the most appropriate to acquire the knowledge that the problem solving needs. We propose a different approach that considers problem solving methods and learning methods on the same ground. This proposal is achieved using a Knowledge Modelling methodology to analyse learning methods. Thus, as the PSM, learning methods may be decomposed into tasks and the methods that solve these tasks. Any task has at least one method that solves it. When a task has as goal the knowledge acquisition, its associated method may be a learning method. Since learning methods have a decomposition in task/methods similar to PSM and can be interleaved freely, problem solving and learning may be integrated in a seamless way.

In section 5 we have analysed some Multistrategy Learning systems. These systems have available several learning methods being useful to deal with complex domains. Some of the Multistrategy Learning Systems are toolkits of learning methods and the user chooses which method has to be used to acquire the necessary knowledge. MTL is a more flexible system that can build new learning methods formed by the combination of some elemental methods. Nevertheless, MTL has a pre-defined order in which the elemental methods are used. Thus, the main shortcoming shown by these systems is the lack of flexibility in choosing the learning method to use. Our proposal is a framework that allows the integration of multiple learning methods without a pre-defined strategy. The multistrategy is a consequence of the KM analysis of a specific application domain since several tasks having as goal the knowledge acquisition can be determined. These tasks can be solved using a learning method. Moreover, since any task can be solved using several methods, knowledge acquisition tasks can also be solved using several learning methods.The flexibility in selecting one of the available learning methods is achieved by means of the lazy problem-centred view already used for selecting PSM.

The next chapter presents our framework for integrating learning and problem solving.

Chapter 3

Framework for Integrated Learning and Problem Solving

1. Introduction

Two phases can be distinguished in the construction of a knowledge system: the *Design* phase (also called Knowledge Engineering phase) and the Problem Solving phase. During the Design phase all the necessary knowledge is acquired, while during the Problem Solving phase the acquired knowledge is used to solve new problems.

In turn, the design of knowledge systems is usually made in two phases: the *knowledge acquisition* phase and the *implementation* phase. Many approaches assume that all the knowledge necessary to solve problems in a domain is acquired during the knowledge acquisition phase. The knowledge modelling (KM) methodologies have been developed to systematise and support this acquisition phase.

Knowledge modelling methodologies present two very different assumptions. On the one hand, most of the KM methodologies produce an analysis of a domain that is implementation-independent. Thus, a new effort is required to operationalise the knowledge into a working representation language during the Implementation phase. On the other hand, they have the assumption that all the knowledge may be acquired before the problem solving phase. This is not necessarily true, due to at least three reasons:

- Some knowledge may be too expensive to acquire from the expert, while this knowledge would be much less expensive to acquire from the experience of actual problem solving in the working task environment. This means that knowledge useful to solve problems would to be acquired in different ways and moments.

- Some knowledge may not be present in the Design phase, nevertheless it may be available in changing the circumstances of the working environment. This means that the acquisition of some knowledge could (and may be should) be delayed until the Problem Solving phase.
- Some KM methodologies associate a single method for each task during the Design phase. Nevertheless, this task-method associations can be difficult to specify during the Design phase. Moreover, a same task can be solved using different methods depending on the problem to solve and the knowledge available. This means that it could be useful to associate several alternative methods to solve a task and delay the selection of a particular one to the Problem Solving phase.

Incremental machine learning (ML) systems are built without these assumptions since they use learning methods to acquire knowledge while solving new problems. ML systems need domain knowledge (usually called *background knowledge*) that is to be acquired somehow. It would be desirable to do so using Knowledge Modelling methodologies, but this is seldom done.

Summarising, a knowledge system dealing with complex domains could benefit from both Machine Learning and Knowledge Acquisition techniques. However, this desideratum has proven to be quite difficult to realise. A main reason for this, in our opinion, is that they use very different representations, i.e. result of Knowledge Acquisition techniques are formulated in high level specification languages (sometimes not implemented, or only partially implemented) and Machine Learning techniques use implementation language structures (like Horn clauses or decision trees).

In this chapter we propose a framework to integrate KM and symbolic Machine Learning. In order to do so the KM methodologies have to be modified to avoid the three assumptions explained above. In more concrete terms, our aim is to provide a framework that fulfils the following goals:

1) To methodologically integrate both Knowledge Acquisition and Machine Learning. This integration will allow the knowledge acquisition in different ways and in different moments (i.e. Design phase or Problem Solving phase).
2) To integrate both learning and problem solving processes. This integration will be made using KM methodologies to analyse and implement learning processes.
3) To select in a lazy problem-centred way the PSM to be used in solving a problem solving task. Each task, during the Design phase, can be

associated with more than one method that can solve it. We propose that the more appropriate method is to be selected during the Problem Solving phase taking into account available information of that specific situation.

4) To integrate several Machine Learning methods into a domain-specific multistrategy learning system. The KM analysis allows the determination of the different learning methods useful to acquire the necessary models.

In the next section we explain the framework we propose and how the four issues above are achieved. Then, in section 3, we describe NOOS, the language used to implement the proposed framework. Finally, in section 4 we illustrate the framework with a short example.

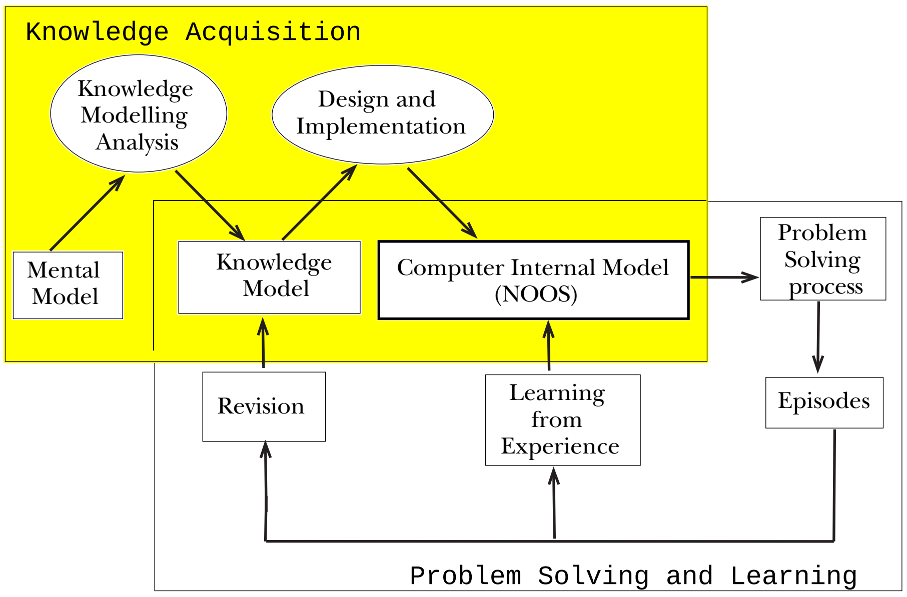

Knowledge Acquisition
Knowledge Modelling Analysis
Design and Implementation
Mental Model
Knowledge Model
Computer Internal Model (NOOS)
Problem Solving process
Revision
Learning from Experience
Episodes
Problem Solving and Learning

Figure 3.1. Scheme of the relationship among Knowledge Acquisition Learning from Experience and Problem Solving (adapted from Aamodt, 1991).

2. Description of the Framework

In this section we describe a framework for integrated learning and problem solving (see figure 3.1) based on the idea that a knowledge modelling analysis permits to make explicit the relation of learning with problem solving. We take a unified approach for inference in learning and problem solving. Also, we propose that a knowledge modelling analysis may be a useful tool for understanding learning, problem solving and their relationship in architectures that integrate both learning and problem solving. Using a knowledge modelling framework for describing both learning and problem solving is useful conceptually but may be very fruitful also at the practical level of building knowledge systems.

In the next sections we first describe the basic elements of the framework. Then we discuss current assumptions of Knowledge Modelling methodologies. Finally, we present how these assumptions are modified in our framework. In (Armengol and Plaza, 1994) and in (Armengol and Plaza, 1995) an analysis of, respectively, CBR and EBL using the proposed framework can be found.

2.1. Basic Elements of the Framework

In section 2 of chapter 2 we have described some KM methodologies such as Generic tasks (Chandrasekaran, 1986), KADS (Wielinga et al., 1992) and Components of Expertise (Steels, 1990). These methodologies agree in determining 1) one (or several) goals to achieve, 2) which is the knowledge necessary to achieve the goal, and 3) how the goal can be achieved. Typically, the necessary knowledge is contained in *models*, and the goals are specified in *tasks*. These methodologies, in addition to goals, also specify how tasks are decomposed in subtasks and which control strategy can be followed. In other words, each task partially specify how can be solved (which subtasks are required). This produces some confusion since some methodologies, such as Generic Tasks, also introduce the concept of *method* as the element that specifies how a task can be solved. Methods are also present in Components of Expertise but with a different meaning: a method is a way in which a model can be acquired.

In our framework, we use the notions of tasks, models and methods. The definition of these elements is the following:

- A *task* is the set of goals to achieve
- A *model* is some knowledge necessary to achieve a task
- A *method* is an inference process that may be used to achieve a task.

The central notion in our framework is that an application domain can be analysed as a task/method decomposition. Each task has associated one or more methods. Each method decomposes a task into subtasks and uses the models to achieve the goal of the task. Notice that the necessary models may be different according to the method used to solve a task. In the next sections each element is explained in detail.

2.1.1. Models

Models contain knowledge to be used to achieve tasks. There are two kinds of models: *problem description models* and *background knowledge models*. Problem description models contain knowledge about a problem to be solved. Background knowledge models contain knowledge that can be used to solve problems. There are four kinds of background knowledge models: `Domain Knowledge`, `Solved Episodes`, `Applicability Conditions`, and `Preferences`. `General Domain Knowledge` models contain specific-domain knowledge that is used by some problem solving method and is obtained by knowledge modelling or by some learning method. `Solved Episode` models contain descriptions of example problems (cases) that either have been solved by the system or have been solved by an external source. In fact, the process of solving a problem can be seen as the construction of a `Solved Episode` model from the problem description model and the background knowledge models.

`Applicability Conditions` models specify, as determined in the Design phase, when each problem solving method (PSM) is to be useful. `Applicability Conditions` models are similar to the *problem solving assumptions* proposed by Benjamins and Pierret-Golbreich (1996), i.e. these models contain a specification of the conditions that have to be satisfied for a method to be useful. Since in our framework a task can be solved using several methods, it is also possible that the applicability conditions of several methods are satisfied, i.e. several methods can be applicable. In such situation the `Preference` models are used.

Finally, `Preference` models, also acquired in the Design phase, define an order of preference upon several PSM if more than one is applicable. Preferences can be static (fixed) or dynamic (inferred from the information available in the specific problem situation).

2.1.2. Methods

Problem solving methods (PSM) embody inference processes that use the knowledge contained in models to achieve some task. In fact, methods in our framework are seen as the process of construction of models from some other models. For instance, a lazy problem solving method is interpreted as the construction of a new `solved episode` model for the current problem from a past `solved episode` model (see section 2 in chapter 4).

There are *elementary* methods and *task decomposition* methods. Elementary methods directly provide a result whereas task decomposition methods decompose a task in subtasks (see figure 3.2). In fact, the result produced by a task decomposition method M associated to a task T can be seen as the combination of the results produced for each subtask in which M decomposes T.

Each method uses some specific models. Moreover, a method has some applicability conditions (contained in the `Applicability Conditions` model), that have to be satisfied for the method to be (possibly) successful. Thus, each method is useful to achieve a specific task while its applicability depends on the available knowledge.

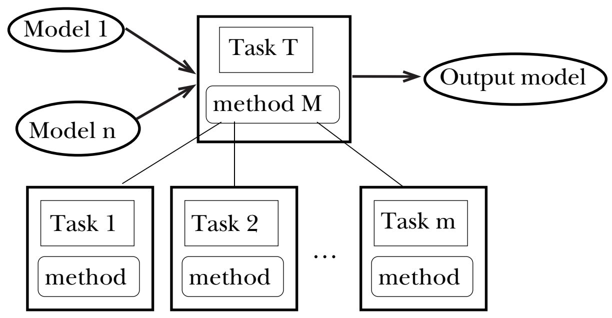


Figure 3.2. Schema of a task decomposition method.

Learning methods are also included in our framework since they are handled as the PSM. There is an special kind of task, the KA-Tasks (see section 2.2) that can be solved using learning methods. A learning method decomposes a KA-Task in subtasks. Learning methods embody an inference process that constructs a specific kind of model from other knowledge sources.

2.1.3. Tasks

Tasks are the elements of our framework that establish the goals (and subgoals) that have to be achieved. Frequently, these goals specify the properties of a model to be constructed. Typically, solving a problem involves constructing the `solved episode` model for that problem (see section 2.1.1).

Each task can be solved using several alternative methods (problem solving methods or learning methods). The appropriate method to solve a task may be selected in a lazy problem-centred way. In other words, the method that can be most useful or efficient to solve a problem task may be selected during the Problem Solving phase according to the knowledge currently available.

This means that some mechanism is necessary to perform this selection. We propose to define tasks at the meta-level (called *ML-Tasks*) for each task T having associated more than one method M_i. Each ML-Task is solved by a meta-level method (called *ML-method*). A ML-Method takes into account the new problem to solve, the PSM applicability conditions and their preferences to select an appropriate method M_i to solve T (see figure 3.3).

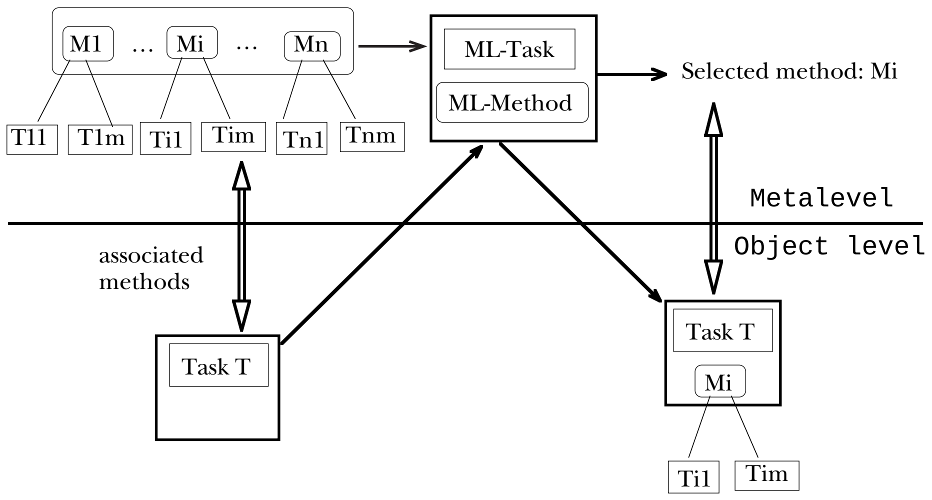


Figure 3.3. A task T having associated several methods needs a ML-Task at the meta-level. This ML-Task uses a ML-method allowing the selection of an appropriate method to solve T.

Thus if, during the Problem Solving phase, a task T can be solved using several methods, their associated ML-Task has to be achieved. This ML-Task is solved by a ML-Method that evaluates the corresponding `Applicability Conditions` models of each method M_i. According to the selected method M_i the task T will be decomposed in different subtasks and can use different models. If several M_i are applicable then the ML-Method also uses the `Preferences` method to select one of them.

2.2. Integration of Knowledge Acquisition and Machine Learning

There are several kinds of knowledge required to solve problems in complex domains (i.e. general domain knowledge, cases, strategies, etc) and each one may be used and acquired in different ways. So, in principle, problem solving in complex domains could use both Knowledge Acquisition and Machine Learning techniques. In fact, both techniques are complementary since they can be applied in different moments:

- *Knowledge Modelling*: All the knowledge is acquired before the Problem Solving phase using knowledge modelling methodologies. However: 1) some knowledge may be more difficult to acquire before the Problem Solving phase, and 2) the results may not have a direct implementation.

- *Machine Learning*: Some knowledge may be acquired during problem solving by a learning method. However: it may be necessary to previously acquire background knowledge. This previous process should be (but usually is not) considered as part of the methodology of Machine Learning. As a consequence, KM methodologies, that could in principle be used, are not used in practice.

Nevertheless, Knowledge Modelling and Machine Learning techniques have a difference that makes difficult their integration: the differences in representation. Knowledge Modelling techniques analyse an application domain and provide a specification of the knowledge in a high level formalism (sometimes not implemented or only partially implemented). Instead, Machine Learning techniques acquire and use representations that are directly used to effectively solve new problems.

Our goal is to provide a framework in which the integration of Knowledge Acquisition and Machine Learning is achieved by a uniform representation. We propose to make this integration by analysing the learning methods using ideas of KM methodologies. Thus, in our framework a learning method can be expressed as a set of tasks, models and methods. In this way, learning methods may be seen (as problem solving methods are seen), in a task/method decomposition perspective (as in figure 3.2).

Thus, learning methods can be viewed as methods associated to tasks whose goal is the acquisition of some specific knowledge. We define a *knowledge acquisition task*, KA-Task, as a task with the goal of acquiring some knowledge that is needed for problem solving but is not directly present or not directly usable. A KA-Task needs to be achieved only when in solving a specific problem another task needs the (non available) knowledge that can be acquired using that KA-Task. Thus, a KA-Task will require to have associated a learning method able to construct the necessary knowledge. This learning method will require, in turn, some input models (that may contain examples and background knowledge) to achieve the KA-Task. Figure 3.4 shows that a task needs a model M1 that has to be acquired by means of a KA-Task. In turn, this KA-Task requires some input models that, using a learning method, produce the model M1.

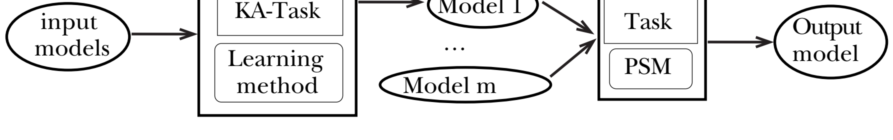


Figure 3.4. Machine Learning and Knowledge Acquisition Integration. The Model1 can be acquired during the Problem Solving phase using a KA-Task that is solved using a learning method.

Our methodological proposal is the following: during the Design phase, by means of a KM analysis, the following steps have to be made:

1. To determine which tasks (and KA-Tasks) have to be solved
2. To determine which models are to be acquired during the Problem Solving phase using KA-Tasks.
3. For each task (and KA-Task) determine which PSM (or learning method) can solve it
4. For each problem solving method (and learning method) determine both the knowledge necessary for the method and the conditions under which the method is applicable.

Thus, for each model that has to be acquired during the Problem Solving phase, the KM analysis will have to define a corresponding KA-Task and determine the learning method that can solve this KA-Task. Also, the KM analysis has to determine 1) the knowledge requirements (models) of each learning method, and 2) under which conditions this knowledge can be acquired. Notice that we are introducing the notion of *delayed knowledge acquisition,* in the sense that some knowledge will be acquired when needed during the Problem Solving phase. We further develop this idea in section 2.3.

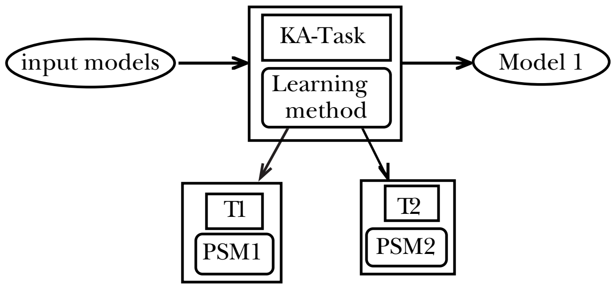


Figure 3.5. Representation of learning and problem solving integration. A learning method decomposes a KA-Task in subtasks. Each subtask can be solved using a PSM.

2.3. Machine Learning and Problem Solving Integration

Knowledge Modelling methodologies assume that all the knowledge is acquired during the Design phase. Our desire is to introduce the possibility to acquire some knowledge during the Problem Solving phase. For this reason, Knowledge Modelling methodologies have to be modified in order to model learning from experience. In other words, in our framework, the acquisition of some knowledge may be delayed until the Problem Solving phase. This notion of delayed knowledge acquisition is the basis for Machine Learning and Problem Solving integration.

During the Design phase, we have to define the tasks, models and methods necessary to solve a problem. In our framework, some of these tasks may be KA-Tasks with associated learning methods. KM methodologies provide a task/method decomposition for each problem solving method (PSM). In the same way, in our framework, a KA-Task can be solved by means of a learning method which can also will be described by a task/method decomposition (see figure 3.5). In turn, each subtask of the learning method may be solved using some PSM.

Summarising, each task in our framework either achieves some goal or acquires some knowledge. Each task can be solved using one o more methods. Each method requires models containing the necessary knowledge to achieve that goal. In particular, models for KA-Tasks may contain examples and background knowledge from which new knowledge can be constructed. In the same way, KA-Tasks have learning methods which can be decomposed into subtasks and each subtask can be solved, in turn, using one or several alternative PSM. Both PSM and learning methods have `applicability conditions` models (such as number of available examples, domain-specific conditions of the problem to solve, etc) that determine which particular method can be applied to solve a specific problem. Thus, the selection of the appropriate method for a task is made in a problem-centred way. In the next section we will explain how to select one method for a task when more than one are available.

Let us suppose that the result of a KM analysis has produced a scheme (as in figure 3.4) in which a task T has associated a PSM. The goal of T is achieved using the models M1 … Mi. These models, except M1, have been acquired during the KA phase. Instead, for M1 only how it can be acquired has been specified. In other words, during the KA phase we have defined a KA-Task having an associated learning method. This learning method (that can in turn be decomposed in subtasks as in figure 3.5) uses some input models and produces as result the model M1 necessary to solve the task T using the associated PSM.

2.4. Problem-centred Selection of Methods

Notions commonly used by KM methodologies are tasks, models and methods. In methodologies as Generic Tasks (Chandrasekaran, 1986) and KADS (Wielinga et al., 1992), each task has associated a unique method. Methodologies as CommonKADS (Wielinga et al., 1993) or Components of Expertise (Steels, 1990) allow the definition of several methods to solve a task. These methodologies assume that during the Design phase there is enough knowledge acquired allowing the engineering team to uniquely specify the appropriate method for each task and subtask. This assumption implies serious limitations whenever a task can be used in different situations where different kinds of knowledge are available or are more efficient. As a consequence, it might be desirable that the method to solve a task can also be changed according to the actual resources.

In our framework, during the Design phase more than one method can be associated to a task. As a consequence, the methodology of Knowledge Modelling has to be changed, removing the single method/task assumption. This new proposal has two implications:

1. The KM analysis may delay the selection of a single method for a task to the Problem Solving phase,
2. During the Design phase the KM analysis has to acquire the knowledge needed in order to select, in a dynamic way, the adequate method for a task taking into account the situation in the task environment.

Let us assume that a task may be solved using several alternative PSM. In order to select one of them we need to acquire, during the Design phase, 1) when each method can be useful (`applicability conditions` model), and 2) how to select only one method if several methods are applicable (`preference` model).

During the Design phase may not be possible to know which of these conditions are satisfied, therefore this evaluation is delayed to the Problem Solving phase. During the Problem Solving phase the applicability conditions of each method are evaluated. Only those PSM whose conditions are satisfied can be applied to solve a task.

The applicability conditions of a method are necessary but not sufficient conditions. This means that a selected method can fail in achieving the pursued goal. This failure can be due to two reasons: 1) the applicability conditions are not accurate enough, 2) the requirements of some (sub)task of the selected method are not satisfied. In both situations another method whose applicability conditions are satisfied has to be selected (using the `preference` model explained in section 2.1.1).

Since the appropriate method to solve each task is selected during the Problem Solving phase according to the available knowledge and the current problem to solve, we say that in our framework method selection is made in a lazy problem-centred way.

In Part III of this work, we describe CHROMA an application that uses lazy problem-centred selection of problem solving methods.

2.5. Multistrategy Learning

Our framework supports the design and implementation of knowledge systems with multistrategy learning. In these knowledge systems learning methods are combined and used in a way that is adapted to each problem solving process as result of the KM analysis of the application domain.

During the Design phase of a knowledge system, several KA-Tasks can be modelled. Each of these KA-Tasks are to be solved by a specific learning method. Thus the resulting knowledge system may include several learning methods following the KM analysis requirements.

Multistrategy learning systems follow two alternative strategies to select the appropriate learning method. One of these strategies is to ask the user for the appropriate learning method to apply. A second strategy is to have a fixed order in which the methods are applied. In our framework the appropriate learning method is dynamically selected in a problem-centred way. Thus, when a task has to be solved, the appropriate PSM is selected according to the new problem to solve. If the selected PSM needs a model that is not available, then this model is acquired using the corresponding KA-Task. This KA-Task has associated a learning method allowing the acquisition of the necessary knowledge. Consequently, the learning performed is adapted to both the domain task and the current task environment since 1) the KM analysis provides the set of tasks, models and methods representing a domain task and, 2) the problem-centred approach that allows the selection of the appropriate tasks, models and methods in the current task environment.

In our framework each KA-Task (as any task) may have associated several learning methods. Each learning method, as any PSM, has (or may have) some applicability conditions. Thus, learning methods can also be selected during the Problem Solving phase in a lazy problem-centred way (as any other PSM). As we have explained in the previous section, the applicability conditions are not sufficient conditions for the applicability of a method (since a method selected according to them can fail). In particular, the applicability conditions of a learning method can be satisfied but may fail to acquire the required model. In such situation a new learning method whose applicability conditions are satisfied has to be selected.

In Part III we describe SPIN, a multistrategy learning system in the domain of marine sponge identification.

3. The NOOS Language

Problem solving in Artificial Intelligence is characterised by a intensive use of specific knowledge about the problems to solve. The goal of the knowledge modelling frameworks is to describe, indepently of the implementation, which knowledge will be used and how it will be used in solving a particular problem. Different knowledge modelling frameworks have proposed different categories of knowledge and different abstractions to describe them.

The NOOS language (Arcos, 1997) proposes a model based in three categories of knowledge: domain knowledge, problem solving knowledge and meta-level knowledge. Moreover, NOOS offers a correspondence from this model to a representation language, based on feature terms, providing a real computational framework for the construction of problem solving systems. Due to its reflective capabilities, NOOS allows a uniform representation of domain knowledge, problem solving methods and learning methods (modelled by means of concepts, relations, tasks, methods and meta-levels).

In the next sections both the model, the feature terms and the NOOS language are briefly described. In (Arcos, 1997) a detailed description of NOOS and its formalisation can be found.

3.1. The NOOS Model

NOOS is based on both the task/method decomposition and the analysis of knowledge requirements for methods. The knowledge modelling in NOOS is related to approaches such as KADS (Akkermans et al., 1993; Wielinga et al., 1993) and Components of Expertise (Steels, 1990). A main difference is that NOOS links subtasks to methods and the methods decompose the tasks, whereas in KADS and in Components the tasks decompose in subtasks and the methods describe how a task can be achieved.

NOOS proposes a model based in three knowledge categories: domain knowledge, problem solving knowledge and meta-level knowledge.

- The *domain knowledge* specifies the set of concepts and the set of relations between relevant concepts of a concrete application.
- The *problem solving knowledge* is modelled by both tasks and methods. *Tasks* represent domain problems that could be solved. *Methods* models the way of how problems can be solved.
- The *meta-level knowledge* (or reflective knowledge) is knowledge about the domain knowledge and the problem solving knowledge. In other words, the meta-level knowledge describes models about concepts, relations, tasks and methods.

There are two kinds of methods: *elementary methods* and *task decomposition methods*. A method is elementary when it solves directly a task. A task decomposition method decomposes a task in subtasks. Subtasks are also solved using methods. To achieve a specific task several alternative methods could be used. This recursive decomposition of tasks in subtasks by means of methods is called *task/method decomposition* (see figure 3.6). Methods can also model relations intensionally described.

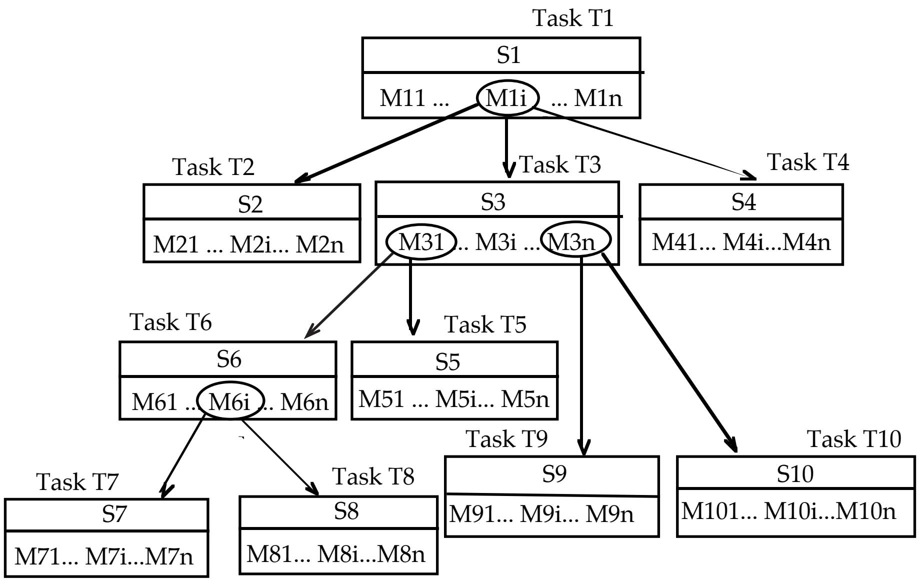


Figure 3.6. Task/method decomposition in NOOS. M_{ij} are PSM solving the task $T_{i.}$ Si is a meta-level method that orders the alternative PSM of the task $T_{i.}$

Concepts, relations, tasks and methods from domain and problem solving knowledge are described at the meta-level knowledge by means of meta-level concepts, meta-level relations, meta-level tasks and meta-level methods respectively. In addition, the meta-level knowledge also includes *preferences* to model the decisions to take about a set of alternatives in both the domain knowledge and the problem solving knowledge. Thus, the meta-level knowledge may be used to model the preference criteria between alternative methods that solve a task. An example of meta-level task is to select a method for a specific task, and another example of meta-level method is to search in memory for methods that can solve a task, select and order some of them according to a set of preferences.

The problem solving process in NOOS is considered as the construction of an *episodic model.* This point of view, the problem solving as modelling, is equivalent to the construction of an episodic model from the input data and from the problem solving knowledge. Thus, the episodic model is composed of the knowledge elements used in solving a concrete problem. Once a problem has been solved, NOOS automatically memorises (stores and indexes) the episodic model of that problem. The episodic memory of NOOS is formed by the collection of episodic models of solved problems. The uniform representation of meta-level knowledge components using concepts, relations, tasks and methods, and the memorisation of episodic models constitutes the basis for the learning integration. The episodic memory is the basic component to integrate learning in NOOS.

This model has been formalised using feature terms, so before describing the NOOS language we will introduce feature terms and several concepts related with them. The feature terms formalism is relevant also for the inductive learning methods we have developed (see Part II).

3.2. Feature Terms in NOOS

Feature terms (also called feature structures or ψ-terms) are a generalisation of first-order terms that have been introduced in theoretical computer science in order to formalise object-oriented capabilities of declarative languages. Feature term formalisms have a family resemblance with, but are different from, unification grammars and description logics (KL-One-like languages) (Aït-Kaci and Podelski, 1993; Carpenter, 1992). The difference between feature terms and first order terms is the following: a first order term, e. g. f(x, g(x,y), z), can be formally described as a tree and a fixed tree traversal order. In other words, parameters are identified by position. The intuition behind a feature term is that it can be described as a labelled graph, i.e. parameters are identified by name (regardless of their order or position).

Given a signature $\Sigma = \langle S, F, \leq \rangle$ (where S is a set of sort symbols that includes $\perp$ and $\top$; F is a set of feature symbols; and $\leq$ is a decidable partial order on S such that $\perp$ is the least element and $\top$ is the greatest element) and a set ϑ of variables, we formally define *feature terms* as an expression of the form:

$$\psi ::= X : s\ [f_1 \doteq \Psi_1 \ldots f_n \doteq \Psi_n\ ] \qquad (1)$$

where X is a variable in ϑ, s is a sort in S, $f_1 \ldots f_n$ are features in F, $n \geq 0$, and each Ψ_i is either a feature term or a set of feature terms. We also identify a feature term with the singleton set of that feature term. Note that when n=0 we are defining only a sorted variable (X : s).

We call the variable X in the above feature term (1) the *root* of ψ, and say that X is *sorted* by the sort s (noted Sort(ψ)) and has features $f_1 \ldots f_n$. A particular example of feature term is the following:

$$\psi_1 = X : \text{person} \begin{bmatrix} \\ \text{last}-\text{name} \doteq \text{Smith} \\ \text{son} \doteq Y : \text{person} \begin{bmatrix} \text{wife} \doteq Z : \text{person} \\ \text{father} \doteq X \\ \text{brother} \doteq T \end{bmatrix} \\ T : \text{person} \begin{bmatrix} \text{father} \doteq X \\ \text{brother} \doteq Y \end{bmatrix} \end{bmatrix}$$

Feature terms provide a way to construct terms embodying partial information about an entity. For instance, the feature term ψ_1 is a partial description of a person. The meaning of the feature term ψ_1 is those individuals that satisfy that partial description; i.e. ψ_1 denotes the subset of individuals such that:

- their `last-name` is Smith
- they have two sons that are persons such that
  - one son has a wife
  - both persons are brothers of each other
  - the father of both persons is the person in the root of the feature term.

A main difference between NOOS and the formalisms presented by Aït-Kaci and Carpenter is that NOOS allows the value of a feature to be a set of values.

The semantic interpretation of feature terms brings an ordering relation among feature terms. We call this ordering relation *subsumption.* The intuitive meaning of subsumption is that of *informational ordering* among partial descriptions. Subsumption in feature terms has the following definition:

- Given two feature terms ψ and ψ', we say that ψ *subsumes* ψ', noted as $\psi \sqsubseteq \psi'$, if there is a total mapping function $\upsilon : \vartheta_{\psi} \rightarrow \vartheta_{\psi'}$ such that:
  1. $\upsilon(\text{Root}(\psi)) = \text{Root}(\psi')$

  and $\forall x \in \vartheta_{\psi}$

  2. $\text{Sort}(x) \leq \text{Sort}(\upsilon(x))$
  3. for every $f_i \in F$ such that $x.f_i \doteq \Phi_i$ is defined, then $\upsilon(x).f_i \doteq \Phi_i'$ is also defined and
     (a) $\forall \psi_k \in \Phi_i, \exists \forall \psi'_k \in \Phi_i'$ such that $\upsilon(\text{Root}(\psi_k)) = \text{Root}(\psi'_k)$ and
     (b) $\forall \psi_k , \psi'_k \in \Phi_i \; (\psi_k \neq \psi_k' \Rightarrow \upsilon(\text{Root}(\psi_k)) \neq \upsilon(\text{Root}(\psi'_k))))$

Intuitively, a feature term ψ subsumes another feature term ψ' ($\psi \sqsubseteq \psi'$) when all information in ψ is also contained in ψ'. For instance, consider the previous presented example of a feature term (ψ_1) and the following one (ψ_2) denoting persons that have married sons:

$$\psi_2 = X : \text{person}\left[\text{son} \doteq Y : \text{person}\left[\text{wife} \doteq Z : \text{person}\right]\right]$$

Clearly $\psi_2 \sqsubseteq \psi_1$, i.e. term ψ_2 subsumes the previous one. Notice that in ψ_2 the `father` feature of person Y is not explicitly given and that X has only one son. Moreover if a term ψ_3 defined as

$$\psi_3 = X : \text{person} \begin{bmatrix} \text{daughter} \doteq W : \text{person} \\ \text{son} \doteq Y : \text{person} \begin{bmatrix} \text{father} \doteq X \\ \text{wife} \doteq Z : \text{person} \end{bmatrix} \end{bmatrix}$$

it is easy to see that ψ_3 satisfies that $\psi_2 \sqsubseteq \psi_3$ but $\psi_3 \not\sqsubseteq \psi_1$.

Feature terms form a partial order by means of the subsumption relationship. From subsumption (equivalent to the *more general than* relation in ML) it is natural to define the operations of unification and anti-unification (AU). In particular, to introduce the anti-unification operation we need to introduce the notion of *equivalence* among feature terms as follows:

- Given two feature terms ψ and ψ' we say that they are *syntactic variants* if and only if $\psi \sqsubseteq \psi'$ and $\psi' \sqsubseteq \psi$.

In other words, two feature terms being syntactic variants are equivalent since they contain the same information. As we will see in the next chapter, we define the anti-unification operation based on the subsumption relationship.

3.2.1. A Brief Description of the NOOS Language

NOOS (Arcos, 1997) is a representation language based on feature terms. Feature terms can be intuitively viewed as data structures similar to records, that contain a set of attributes (features). Feature terms allows the representation of incomplete knowledge (Aït-Kaci and Podelski, 1993). Incomplete knowledge arises two problems: the so-called problem of *unknown values* in Machine Learning and the problem of *irrelevant attributes* that is specially important in attribute-value representation.

Concepts are represented in NOOS as feature terms, and relations are represented as features. In particular, a feature term representing a concept contains the set of features of that concept. The syntax to build feature terms is based on lists beginning by the token `define`. For instance, the following feature term is the definition of the concept *typical-person* described by two features: `last-name` and `age`.

```
(define TYPICAL-PERSON
   (LAST-NAME Smith)
   (AGE 30))
```

The notion of refinement is introduced in the NOOS language as a mechanism to construct feature terms from other feature terms. Refinement has two different aspects: the *reusability* of the code and the *sort/subsort hierarchies.*

The reusability of the code is made using refinements. In fact, a new feature term is always constructed as a refinement of a existing feature term. In other words, if a new term N is defined as a *refinement* of a existing term E, N will include all the features defined in E that have not been redefined in N. For example, the definition

```
(define (typical-Person TYPICAL-FEMALE)
   (LAST-NAME Taylor)
   (HAIR blonde))
```

represents a new feature term called *typical-female* that is a refinement of the feature term called *typical-person*. This means that *typical-female* shares with the *typical-person* feature term the features `age` and `last-name`. Because the `age` feature does not appear in *typical-female* it is reused so the value of `age` in this feature term is 30 as in *typical-person*. Instead, the `last-name` feature is redefined in *typical-female* and so it has a different value than in *typical-person*. Moreover, the *typical-female* feature term has a new feature called `hair` with value *blonde*.

Refinement is also used to define the sort/subsort hierarchy. In the previous example we can consider that *typical-female* is a subsort of the sort *typical-person*.

When the name of the refinement feature term is not specified, the feature term is called *anonymous*. Anonymous feature terms are also defined as feature values but without defining a new sort. For example, the following feature term called *typical-male* is a refinement of the *typical-person* term:

```
(define (typical-person TYPICAL-MALE)
   (WIFE (define (typical-female)
            (DAUGHTER (define (typical-female)
                         (TALL 1,80))))))
```

The feature term *typical-male* reuses the features `age` and `last-name` of *typical-person* (with values *30* and *Smith* respectively). The feature `wife` defined in the *typical-male* feature term has as value an anonymous feature term of the sort *typical-female*. In turn, this anonymous feature term reuses the features `age` and `last-name` of *typical-female* and has an additional feature called `daughter`. The value of the `daughter` feature is also an anonymous feature term of the sort *typical-female*. This means that the `daughter` of the `wife` of a *typical-male* is *blonde* and her `last-name` is *Taylor*. Moreover, this *typical-female* is 1,80 m tall.

In our example, the sort/subsort hierarchy defined by the refinements defines two relations among sorts: *typical-person* ≤ *typical-male* and *typical-person* ≤ *typical-female*. In turn, *typical-male* and *typical-female* can also be used in a new refinement to define new subsorts.

NOOS provides an initial set of sorts with an order relation among them. The feature term called *any* represents the minimum information from which all other feature terms are defined by refinement, i.e. all the feature terms are refinements of *any*. A feature term defined as a refinement of *any* may be defined as follow:

```
(define PERSON)
```

that is equivalent to `(define (any PERSON))`. For instance, the definition of *typical-person* above is also a refinement of *any*.

Thus, the refinement notion is a crucial mechanism in NOOS for constructing feature terms. This mechanism involves two aspects: 1) the construction of feature terms by reusing other feature terms, and 2) the definition of a sort hierarchy.

NOOS provides two kinds of references: *name references* and *path references*. A name reference is when the value of a feature is the name of a feature term. For example, the value *blonde* of the feature `hair` in the definition of *typical-female* above. Path references are used to refer anonymous feature terms (which have no name). There are two kinds of path references: absolute and relative. An *absolute path reference* is a list that starts with the token '`>>`' followed by a sequence of feature names, then the '`of`' token, and finally the name of a named feature term. For example, the path reference to the feature `tall` of the feature term *typical-male* above is the following:

```
(>> wife daughter tall of typical-male)
```

Relative path references are as path references that elide the name of the feature term, i.e. they only specify a sequence of feature names. A relative path reference is bound to a specific description by the rules of scope and refinement. The scope of NOOS is lexical, since a relative reference is determined by the text in which it appears. Specifically, a relative path reference is bound to the root of the description in which it textually appears (the outmost '`define`' in the text where it occurs).

Both kinds of path references, absolute and relative, represent *path equality* in feature terms. For example in the following definition of the *Mary* feature term:

```
(define (typical-female MARY)
   (HUSBAND James)
   (HAIR black)
   (SON (define (typical-male)
          (FATHER (>> husband)))))
```

the relative path reference `(>> husband)` in the feature `father` of the son of Mary refers to the `husband` feature of *Mary* (the root). This means that the value *James* can be found following two paths: `(>> husband of Mary)` and `(>> father son of Mary)`. That is to say, a path reference establishes an equality between two paths: the path reference `(>> husband of Mary)` and the path between the root and the point where that path reference occurs, namely `(>> father son of Mary)`. In figure 3.6 a graphical representation of this path equality is shown.

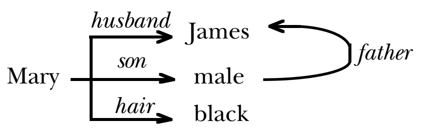


Figure 3.7. Representation of a feature term using a labelled graph.

In addition to the `define` construct to build feature terms, feature terms in NOOS can be described by labelled graphs. Nodes of these graphs are labelled with sorts and edges are labelled with named parameters (called *features*). Figure 3.7 shows the representation of the example *Mary* as a labelled graph. Notice that the path equality is represented as node sharing. NOOS provides a graphical user interface (*browser*) to represent labelled graphs. In figure 3.8 there is the browser of the *Mary* feature term. We will use indistinctly the three feature terms representation: record-like representation (as expression (1) in section 3.2), textual descriptions using `define`, and graphical displays using browsers.

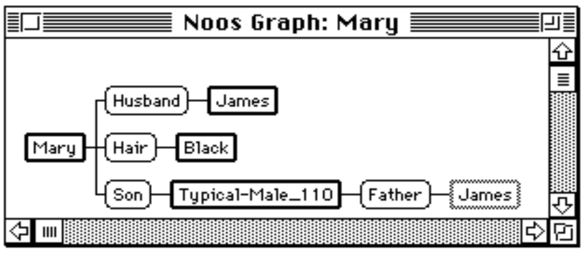


Figure 3.8. Browser of the feature term *Mary*. Nodes in grey (like James) represent path equality.

Methods in NOOS are represented as evaluable feature terms. The features of a method are subtasks and/or models. Thus, the collection of features defined in the description of a method are interpreted 1) as the decomposition of the method into subtasks, or 2) as the models that are necessary by the method. This task decomposition allows the definition of (sub)methods for each subtask. The value returned by a method can be any feature term, including a method. The NOOS language provides a set of basic methods, called *built-in methods*, from which new methods can be constructed by refinement and/or combination of them. Examples of NOOS built-in methods are arithmetic operations, set operations, logic operations, operations for comparing feature terms and other basic constructs such as conditional or sequencing. For example we can define the method `red-element` as a refinement of the *built-in* `conditional-method` as follows:

```
(define (conditional-method RED-ELEMENT)
   (ELEMENT )
   ((CONDITION (DEFINE (identity?)
                (ITEM1 red)
                (ITEM2 (>> colour element)))))
   (RESULT true)
   (OTHERWISE false))
```

Let us suppose that we want to classify some objects as belonging to a class. Elements of this class have the colour red. Thus, the method above, given an element, tests if it has the colour red. If the element is red the method returns *true*, otherwise returns *false*. The value of the `condition` feature is inferred using a *closed method*. Closed methods are represented by a double parenthesis (as shown in the feature `condition` of the *red-element* feature term). By means of a closed method the value of a feature is described by an inference instead of a fixed value.

We say that a feature F of a feature term ψ is *reduced* when the closed method or the path reference of F has been evaluated yielding a value for F. Also, we say that a feature term ψ is in *normal form* when all their features are reduced.

Inference in NOOS is on-demand. Inference starts when a user asks to solve a specific task by means of a query expression that engages a particular task. Path references can be used as query expressions, for instance `(>> age of` *`Mary`*`)`. Since methods are decomposed into subtasks, when a method is evaluated its subtasks are also engaged and their respective methods are also evaluated. Therefore, the inference process in NOOS can be viewed as a chaining process along the task/method decomposition tree. This recursive chaining finishes when a method directly uses factual knowledge. A task is achieved when its corresponding method is successful and a method is successful when all its tasks are achieved.

Reflection in NOOS is impasse-driven. When a task has to be solved, two types of impasses can occur: (1) there is no method specified for solve the task, and (2) there are several alternative methods able to solve the task. If there is no method associated to the task, the control of the inference is passed to the corresponding task at the meta-level. A meta-level task can also have associated a method to achieve it. Typically, such a meta-level method infers a partially ordered set of alternative methods for the current task. Thus, the result is equivalent to impasse (2). In this situation of multiple method impasse one method is reflected down at the base level using preferences among methods. These preferences will be introduced in section 3.2.3.

3.2.2. Episodic Memory

NOOS automatically stores decisions taken during the inference process. The set of these decisions constitutes the *episodic memory* of the system. In our framework the episodic memory corresponds to the `solved episodes` models described in section 2.1.1. The *episodic model* of a problem is the instantiation of the task/method decomposition used in solving that problem. In other words, the *episodic model* of a problem is the new `solved episode` model constructed from the `problem description` model by the PSM used.

NOOS also supports an *introspection* capability to incorporate learning mechanisms. In other words, NOOS provides a way to reuse the experience acquired in solving problems in order to solve new problems.

Introspection is able to examine the contents of the episodic memory, and NOOS provides two introspection forms: access by path and access by contents. *Access by path* is performed by combining reflective operations and path references. This kind of access provides a way to access specific portions of the episodic memory. *Access by contents* is performed by retrieval methods that allow to retrieve previous similar episodes from the episodic memory using similarity (relevance) criteria. Retrieval methods are necessary for knowledge systems that need to retrieve episodes in which a concrete task has been achieved using facts and features similar to the current problem. CBR methods can be analysed and implemented in this way (see chapter 4).

Retrieval methods are a subset of the *built-in* methods provided by NOOS. Moreover, we can design for specific applications new retrieval methods by refinement and combination of the existing ones. Using retrieval methods, previous similar episodes can be accessed, analysed, and finally reused. Similitude criteria are determined by specific knowledge about the relevance of several features or about the requirements of the problem solving methods. It is also possible to learn relevances (or similarities) of features (see chapter 7).

3.2.3. Preferences

The reasoning based on preferences is used in NOOS to model the decisions taken about a set of alternatives present in both the domain knowledge and the problem solving knowledge. Preferences in NOOS are represented as partially ordered set, which are defined as pairs $<A, \prec>$ where A is a set of alternatives and $\prec$ is a binary relation that is reflexive and transitive on A. Thus, preference-based reasoning allows the construction of partial orderings of these alternatives.

Knowledge about preferences is described in NOOS by means of *preference methods*. There are two kinds of preference methods: methods for constructing preferences and methods for combining preferences. Preference construction methods are useful to create partial orderings of sets using some domain specific criterion. Preference combination methods are useful to create new orderings from two partially ordered sets already obtained from the application of either a construction method or another combination method.

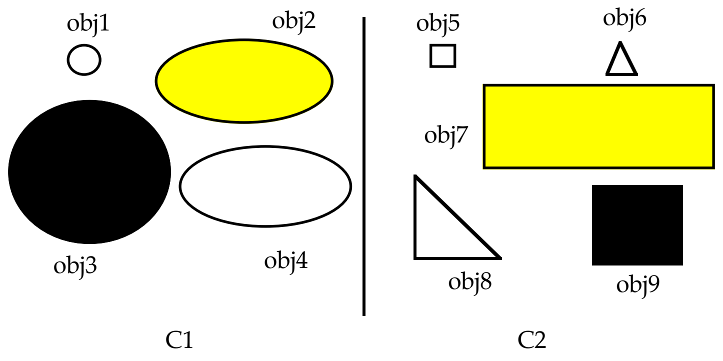
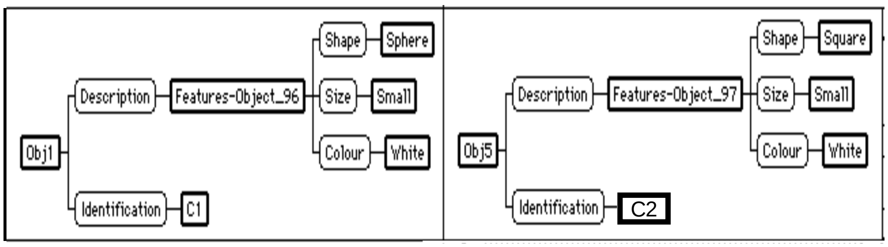


Figure 3.9. A domain example.

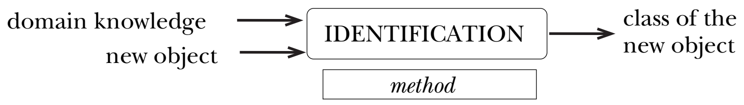
4. An Example

In this section we illustrate with a very simple example how problem solving can be modelled using our framework and how this modelling can be implemented using the NOOS language.

Let us suppose the domain of the objects in figure 3.9. In this domain, each `solved episode` model contains an object represented by a feature term of the *object* sort having two features: `description` and `identification`. The value of `description` is an object belonging to the *features-object* sort that is described by three features: `shape`, `size`, and `colour`. The `identification` feature contains the solution class to which the described object belongs. In particular, objects can be classified as belonging to two classes: *C1* and *C2*. Thus, for instance, objects Obj1 and Obj5 in figure 3.9 are shown in the following browser:



Let us suppose that our goal is the identification of new objects as belonging to one of the two classes. This problem can be modelled in the following way:



Let us assume that the KM analysis of this domain specifies two ways (methods) in which the `identification` task can be solved:

A) The only available domain knowledge are the classified objects in figure 3.9. A new object may be classified according to its similitude with some of these classified objects.

B) The description of each class is available. In this situation the new object can be classified as belonging to the class whose description satisfies.

We will also assume that we do not know which are the descriptions of the classes *C1* and *C2* shown in figure 3.9. During the KM analysis we can decide that these description will be acquired during the Problem Solving phase using a learning method. If so, during the KA phase we have to define a KA-Task having as goal to acquire the descriptions of these classes. This KA-Task can be solved using an inductive learning method that constructs the class descriptions from the classified examples, as we will show later in this section.

In the rest of this section we use the framework to model this simple task and the NOOS language is used to illustrate the implementation. We will suppose that the new object ObjN that has to be identified is the following:

```
(define (object ObjN)
   (DESCRIPTION (define (features-object)
                   (SHAPE circular)
                   (SIZE small)
                   (COLOUR stripped))))
```

4.1. CBR Methods

In this section we develop the modelling of the `identification` task solved using the method A above that is, in fact, a CBR method. The CBR method needs to search the memory of cases for some object similar to the new one. In our domain, we search for some object in figure 3.9 similar to ObjN. Let us suppose that in this domain `size` is the most important feature in order to compare two objects. If the `size` feature is not discriminant enough, the next feature that is important is the `shape`. Now we can proceed to define a CBR method that solves the proposed problem[1].

[1] We may not have the knowledge of which features are more important. This more complex situation is addressed by the LID method (explained in chapter 7) where learning involves determining which are the most important features.

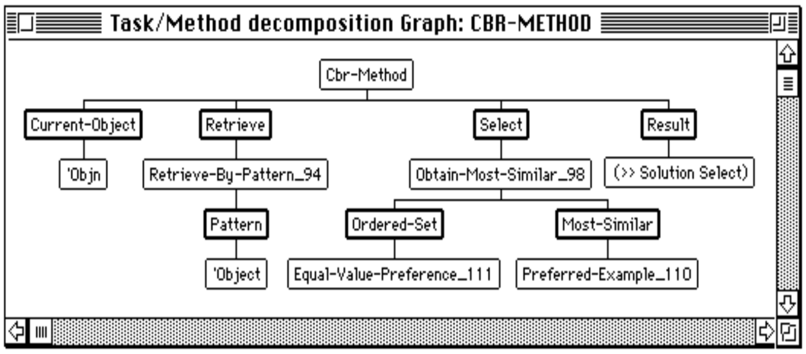


Figure 3.10. Browser of the `CBR-method` associated to `identification` task.

A CBR method can be decomposed in three tasks: `retrieve`, `select` and `result`. NOOS allows the representation of CBR as a method with the three tasks represented as features (as shown in figure 3.10). Because the problem above is a classification problem, the `result` task consists simply of taking for the current problem the same class as that of the best precedent. From the solution provided by `result` task and the description of the current problem a new `solved episode` model is constructed. This new `solved episode` model may be used to solve further problems.

`Current-object` is a problem description model containing the description of the object to identify (in our example, ObjN). The `retrieve` task uses the NOOS *built-in* method called `retrieve-by-pattern`. This method has a feature (called `pattern`) whose value is a feature term (`object`, in our example, that will be explained below). The feature term in the `pattern` feature represents the minimum information that a training example has to contain in order to be retrieved. In other words, `retrieve-by-pattern` method returns as result the set X of training examples such that `pattern` $\sqsubseteq$ X. In our example, the `retrieve` task is implemented as follows:

```
(define (retrieve-by-pattern RETRIEVE)
   (CURRENT-OBJECT object)
   (PATTERN (define (object)
            (DESCRIPTION (define (features-object)
                         (SIZE (>> size description current-object)))))))
```

That is to say, `retrieve-by-pattern` will retrieve all the objects having the same `size` that the object to identify (`current-object`). Notice that the pattern depends on the object to identify since the value of the `size` feature changes according to each `current object`. In our example, the `current-object` is ObjN, so we can define

```
(define (retrieve RETRIEVE-ObjN)
   (CURRENT-OBJECT ObjN))
```

Since `retrieve-ObjN` is a refinement of `retrieve`, the `pattern` feature is a feature term belonging to the sort *object*. The `description` feature of this object has the `size` feature with the value of the ObjN `size` feature. The ObjN size is *small*, therefore `retrieve-ObjN` retrieves all the objects having small size, i.e. C = {obj1, obj5, obj6}.

Next, the `select` task has the goal of extract only one element of C using a method, called `obtain-most-similar`, that decomposes in two tasks (see figure 3.10): `ordered-set` and `most-similar`. The goal of `ordered-set` task is to order C according to some criteria. A possible ordering criterion is to prefer those objects having the same `shape` as ObjN. Then, `most-similar` task selects the element of C most preferred according to the last task. In our example, the object finally retrieved and selected is Obj1, since it is the only object having the same size and the same shape that ObjN.

Finally, the `result` task inspects the `solved episode` model containing the selected object and retrieves the class to which it belongs. In our example, Obj1 belongs to *C1*, thus ObjN will be identified as belonging to *C1*. So, the value for the `identification` feature of the ObjN is *C1*. Therefore, the new solved episode will contain the following object:

```
(define (object ObjN)
   (DESCRIPTION (define (features-object)
                    (SHAPE circular)
                    (SIZE small)
                    (COLOUR stripped)))
   (IDENTIFICATION C1))
```

4.2. Classification Method

In this section we develop the modelling of the `identification` task solved using the B method (see introduction of section 4), that we call now `classification-method`. Input models of `classification-method` are `current-object` and `description-classes` (see figure 3.11). During the KA phase the descriptions of the classes *C1* and *C2* are not available but we can define a KA-Task that computes them during the Problem Solving phase. This KA-Task has associated a learning method, called `induction`. We will explain this KA-Task in next section. Now we assume that the descriptions of *C1* and *C2* are the following:

$$
\begin{aligned}
C1 = {} & X: \text{object}\left[\text{shape} \doteq \text{circular}\right] \\
& \vee \\
& Y: \text{object}\left[\text{shape} \doteq \text{oval}\right] \\[1em]
C2 = {} & Z ; \text{object}\left[\text{shape} \doteq \text{triangle}\right] \\
& \vee \\
& V: \text{object}\left[\text{shape} \doteq \text{rectangle}\right] \\
& \vee \\
& W: \text{object}\left[\text{shape} \doteq \text{square}\right]
\end{aligned}
\qquad \text{(A)}
$$

Thus, the `classification-method` has only one task (see figure 3.11): `identify-object` task. The goal of `identify-object` task is to determine in which class the current object (ObjN) may be classified. The `identify-object` task has associated a method based on subsumption, called `identify-by-subsumption`. This method searches for all the classes such that their description subsumes the description of the current object. In our example only the description X of *C1* subsumes the description of ObjN, therefore ObjN is classified as belonging to the class *C1*. So, a new `solved episode` model is constructed containing the object ObjN which `identification` feature has value *C1*.

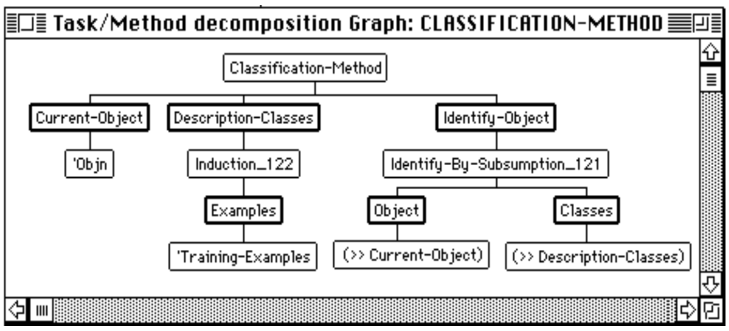


Figure 3.11. A browser of the `classification-method`.

4.3. Induction method

In the previous section we have described the `identification` task having associated a method (`classification-method`) that needs two models: `current-object` and `description-classes`. The `description-classes` model is not available, therefore it can be viewed as a KA-Task that has as goal to induce a description for each solution class from the training examples.

Let E = {Obj1 ... Obj9} be the set of `solved episode` models (those in figure 3.9), and {*C1, C2*} the solution classes to which the described objects can belong. Figure 3.12 shows the task/method decomposition of the KA-Task `description-classes`. Given the set E and a solution class C_i the goal of this KA-Task is to obtain a description for C_i. This KA-Task can be solved using a learning method that decomposes in three subtasks: `positive-examples`, `negative-examples` and `build-description`.

The `positive-examples` task has as input models the set of `solved episode` models E and the class C_i of which the description has to be obtained. The output model of this task is the set E^+ of `solved episode` models describing objects that belong to C_i. The `negative-examples` task builds a model E^- containing the `solved episode` models that have not been included in E^+.

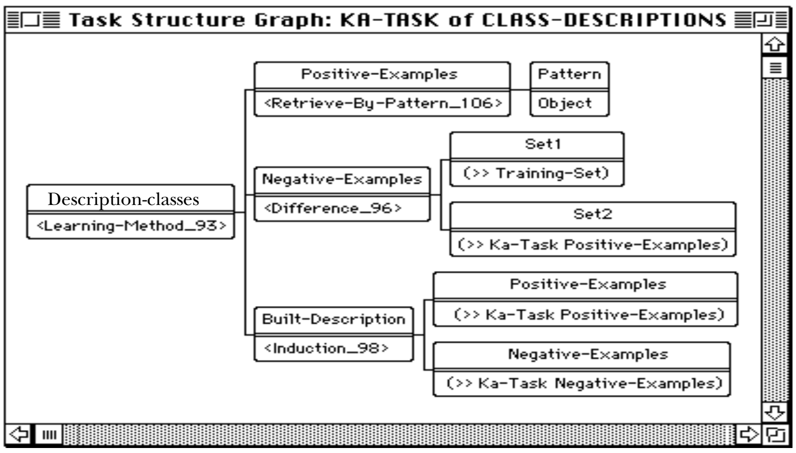


Figure 3.12. Task/Method decomposition of the KA-Task that obtains the description for a solution class.

For instance, the `solved episode` model containing the description of Obj6 in figure 3.9 is the following:

```
(define (object Obj6)
   (DESCRIPTION (define (features-object)
                  (SHAPE triangular)
                  (SIZE small)
                  (COLOUR white)))
   (IDENTIFICATION C2))
```

Feature `description` contains the problem description of an object and the `identification` feature contains the solution class to which it belongs. Thus, both `positive-examples` task and `negative-examples` task define an extensional model of C_i (i.e. the set of positive examples E^+ and the set of negative examples E^-). The method used to solve the `positive-examples` task is the NOOS *built-in* called `retrieve-by-pattern`. For instance, the positive examples of the class *C1*, namely E^+, are those examples having in the feature `identification` the value *C1*. The remaining training examples, i.e. $E^- = E - E^+$, are considered as negative examples of *C1*.

Then the `built-description` task uses both models E^+ and E^-, and an inductive learning method to build a model with the description D_i for the class C_i. In our example the result of this task are the descriptions labelled as (A) in section 4.2. In Part II we explain some of the inductive learning methods based on feature term anti-unification that can be used to compute the class descriptions.

4.4. Lazy Selection of Methods

During the KM analysis we have associated two methods to solve the `identification` task: `CBR-method` or `classification-method`. Now, knowledge modelling has also to define both the applicability conditions of the methods and the preferences between them. In other words, during the KM analysis, the knowledge for selecting one method has to be acquired. In order to select one method for the `identification` task, we will have a task (that we call `ML-identification`) that will be at the meta-level. This `ML-identification` task has associated a meta-level method (that we call `ML-Selection` method) that selects the appropriate method for `identification` task in a lazy problem-centred way.

Let us assume that the KM analysis has determined that the application of the `classification-method` is more efficient than applying the `CBR-method`. Therefore, we define a sequential strategy consisting of first applying the `classification` method and, if it fails, the solution is obtained using `CBR-method`. This sequential strategy can be represented in NOOS as follows:

```
(define OBJECT
   (DESCRIPTION (define (features-object)))
   ((IDENTIFICATION (define (classification-method)
                          (CURRENT-OBJECT (>> description))
                          (EXAMPLES training-set))
                    (define (CBR-method)
                          (CURRENT-OBJECT (>> description))))))
```

Using this definition, when an object has no value in the `identification` feature, the `classification` method will be used to obtain a value. If this method fails then the `CBR-method` will be used.

Let us suppose now that the following object has to be identified:

```
(define (object ObjM)
   (DESCRIPTION (define (features-object)
                     (SHAPE trapezoid)
                     (SIZE medium)
                     (COLOUR stripped))))
```

Using the sequential strategy described above, none of the class descriptions subsume ObjM, thus `classification-method` fails. After this failure, the `CBR-method` is selected and applied. The `CBR-method` retrieves precedents according to its `size`, therefore two objects are retrieved (Obj8 and Obj9) since both have *medium* size as ObjM. The `selection` task of the `CBR-method` selects the object having the same `shape` that ObjM, although in this example none of the retrieved precedents have *trapezoid* shape (the ObjM `shape`). As consequence both precedents Obj8 and Obj9 are selected. Nevertheless, because both, Obj8 and Obj9, belong to the class *C2*, ObjM is also classified as belonging to the class *C2*.

A more sophisticated strategy for selecting the appropriate method could be elaborated during the KM analysis. For instance, when the `shape` of the new example has not appeared in any example, the `classification` method will always fail. Consequently, the `identification` task can only be solved using the `CBR-Method`. Thus, the selection of the appropriate method could be made dynamically. This dynamic strategy may be implemented solving the `ML-identification` task at the meta-level of `identification` task using the `ML-selection` method show in figure 3.13. The `ML-selection` method has been implemented using the NOOS *built-in* called `conditional-method`. The condition of `ML-selection` method checks if the `shape` of ObjM is already present in some `solved episode` model. If this is the case, then `classification-method` is selected by the `selection` task at the meta-level. As before, if the `classification-method` eventually fails, then the `CBR-method` is selected as second option. On the other hand, when the `shape` of the new object (ObjM) is not present in any `solved episode` model, `ML-selection` method chooses the `CBR-method` only. This example we have presented is intentionally simple but it shows how introspective capabilities allow a dynamic adaptation of multistrategy learning systems to the knowledge modelling analysis of the domain. Chapter 8 shows a similar dynamic strategy that has been used in the much more complex real world CHROMA application.

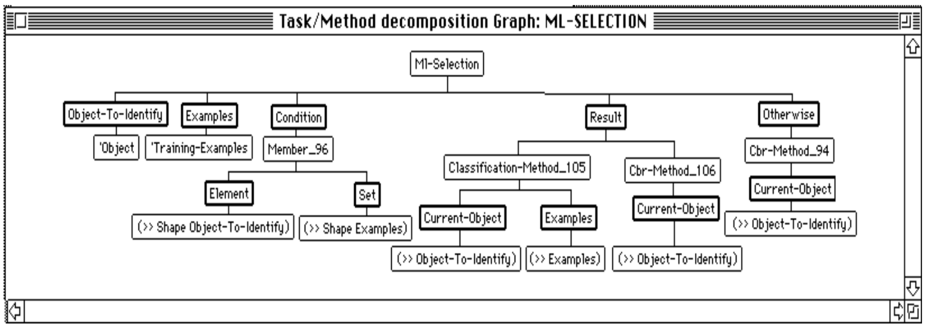


Figure 3.13. Task/method decomposition of the `ML-selection` method used at the `identification` task meta-level.

5. Conclusions

KM methodologies assume that all the knowledge can be acquired during the Knowledge Acquisition phase. Instead, Machine Learning methods assume that most of knowledge can be automatically acquired from some background knowledge during the Problem Solving phase. Commonly, this background knowledge is acquired without using a concrete methodology. We think that the integration of both Knowledge Acquisition methodology and Machine Learning techniques may be useful. Nevertheless, the integration of Machine Learning and Knowledge Acquisition has two main issues:

1) KM methodologies have to be extended in order to acquire knowledge during the Problem Solving phase.
2) Both Machine Learning and Knowledge Acquisition use different kinds of representations.

In this chapter we have introduced a framework addressing both issues. In this framework the integration of both Knowledge Acquisition and

Machine Learning is based on a knowledge modelling approach to knowledge acquisition and learning. In other words, we propose to analyse learning as on inference process by means of KM.

To address the first issue above, we propose a new kind of tasks, called KA-Tasks, whose goal is the (delayed) acquisition of some knowledge required for a PSM. KA-Tasks have associated learning methods to achieve its goal. In this way, we can delay the acquisition of some knowledge to the Problem Solving phase.

The issue of difference in representations is addressed by using the NOOS representation language and the underlying formalism of the feature terms. This formalism provides some of the essential characteristics of the NOOS language, such as

1) The Knowledge Modelling analysis of the domain can be easily represented in the proposed framework. Elements of our framework are all them represented using feature terms.

2) Learning is viewed in NOOS as a search in the feature terms space. Machine Learning can be implemented in our framework using some NOOS capabilities such as retrieval and introspection.

Using our framework, a knowledge system is modelled by a task/method decomposition. In this decomposition, each task can be solved using several alternative methods. Each method is applicable under some conditions (defined in `applicability conditions` models) and some preferences may have been established for selecting among alternative PSMs.

This same analysis can be made for the KA-Tasks, i.e. a KA-Task is solved using a learning method that can be decomposed into subtasks. In turn, each subtask can be solved using some problem solving method. As a consequence, both problem solving and learning can be integrated in a natural way during the knowledge system performance.

An advantage of the proposed framework is that it allows a seamless integration of learning and problem solving. During the KM analysis we can determine which are the models required by each PSM and which learning methods can be used to acquire some of these models. A second advantage is that the selection of a PSM for a task can be delayed until a problem has to be solved. This selection is made according to situation-specific criteria for the problem, applicability conditions of PSM and preferences in using a PSM.

Commonly, learning methods handle either attribute-value or relational formalisms. The use of a new formalism such as feature terms has bound us to define new learning methods capable to deal with them. These new methods are introduced in Part II.

PART II

MOTIVATION

In the previous chapter we have introduced a framework that, among several capabilities, allows the integration of learning and problem solving. Into this framework we have defined the KA-Tasks that are tasks whose goal is the acquisition of knowledge needed by a problem solving activity and that are solved using learning methods.

The proposed framework is representable using the NOOS language, the basic formalism of which are feature terms. The feature terms formalism allows the integration of Knowledge Modelling and Machine Learning techniques. Thus, if is necessary the design of learning methods capable to deal with feature terms.

In Part II we provide, in chapter 4, a brief introduction of some concepts necessary to define new learning methods that handle feature terms. Then, in chapter 5 we describe INDIE a bottom-up inductive learning method. Chapter 6 describes DISC a top-down inductive learning method. Chapter 7 describes a lazy learning method, LID, that estimates similitude among feature terms.

Chapter 4

Learning Methods using Feature Terms

The use of feature terms as representation formalism requires to define new learning methods capable to deal with them. In this chapter we explain how both inductive learning and lazy learning techniques can be used into the space of feature terms. In our framework (and in NOOS), learning techniques will be considered as methods. In chapters 5 and 6 we will describe two inductive learning methods and in chapter 7 we will describe a lazy learning method for feature terms.

1. Inductive Learning Methods using Feature Terms

Inductive learning methods can be defined as those methods that systematically produce intensional concept descriptions from extensional concept descriptions. In other words, from the specific knowledge provided by domain examples, an inductive learning method is capable to obtain general domain knowledge.

In our framework, the goal of an inductive learning method is to generate the knowledge needed by a problem solving method (PSM). Thus, we consider induction as a method (see figure 4.1) that, using as input models both the `solved episodes` models and the `background knowledge` models, is capable to generate a new model (`domain theory` model) useful for a PSM to solve a specific kind of problems. The application of a PSM is viewed as the process of construction of a model (called `new solved episode` model) that contains information relative to the problem solving process of a new problem.

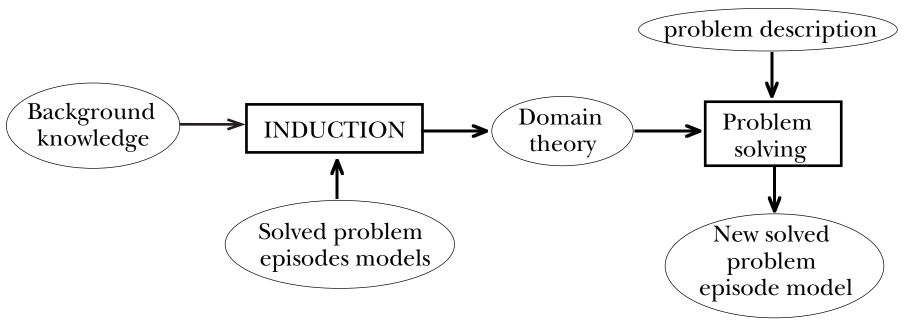


Figure 4.1. Scheme representing an inductive learning method.

A typical use of an inductive method is the generation of a class description for a category or concept from a set of examples. The acquired knowledge will be used by a method (*problem solving* in the figure above) to decide whether or not new examples pertain to a certain category.

There are two families of inductive learning methods. One of them, the family of propositional learners, includes algorithms such as ID3 (Quinlan, 1986) or C4.5 (Quinlan, 1993), requires that domain objects are represented as attribute-value pairs. The other family are the relational techniques, including FOIL (Quinlan, 1990) and the Inductive Logic Programming (ILP) systems that handle examples and domain knowledge represented as Horn clauses. For some tasks, the representation of the domain objects using a structured representation is more natural. Structured representation means that an object is represented by a set of attributes which values may be, in turn, objects with a set of attributes. Feature terms is a formal view for the structured representation of objects.

Inductive methods can be characterised as search methods over a hypothesis space (Mitchell, 1982). The search techniques usually comply to certain biases: constraints upon the hypothesis space effectively searched and strategies for searching certain subspaces before others. These biases of learning methods are similar to assumptions for problem solving methods (Benjamins and Pierret-Golbreich, 1996). For instance, a learning inductive method can be exhaustive (or complete if it assures it will find a generalisation if it exists) or not exhaustive. In section 3.1.2 of chapter 2 we have seen that relational learners structure the hypothesis space according to the notions of generalisation and specialisation. To constraint this hypothesis space relational learners introduce a partial order between hypotheses (e.g. in Horn clauses it is called θ-subsumption). Then, this ordered space is searched according to the bias of each inductive method. In our framework, feature terms offer a representation formalism that is a subset of first order logic where subsumption provides a well defined and natural way for defining generalisation relationships: a feature term ψ is more general than (or equal to) another feature term ψ' if and only if ψ subsumes ψ' ($\psi \sqsubseteq \psi'$).

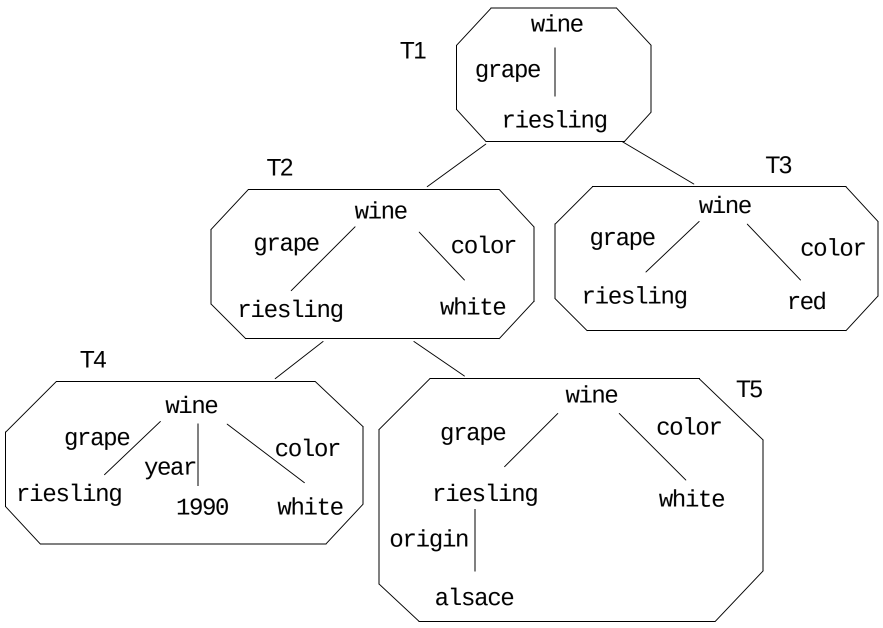


Figure 4.2. Example of lattice formed using the subsumption among feature terms.

Summarising, the inductive learning methods that we have developed use the same representation language to represent both examples and generalisations. That is to say, examples and generalisations are uniformly represented using feature terms, so both are considered partial descriptions and have a structured representation. The *more-general-than* relation commonly used in ML is here the subsumption relation among feature terms. When we say that a description D subsumes an example e_i (or a description d_j), this is equivalent to say that D is more general than e_i (or d_j). The feature term T1 in figure 4.2 subsumes all other terms, i.e. it is more general than others (T2, T3, T4 and T5). Also, the feature term T2 subsumes T4 and T5, whereas T3 does not subsumes any term.

The anti-unification of two examples e_1 and e_2 is a description that is also a term in the formalism, and is a most specific generalisation subsuming the examples e_1 and e_2. In our example of figure 4.2, T2 is the anti-unification of the feature terms T4 and T5 (see next section).

Induction from a set of examples means to obtain a description generalising those examples. If the examples have a structured representation (feature terms), induction means to search for a term subsuming the examples represented as terms. Thus, if T3, T4, and T5 are considered examples, the feature term T1 can be viewed as the generalisation of them.

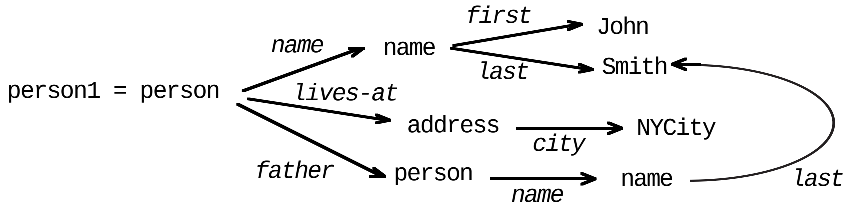
Inductive learning methods can work either on positive examples only or on positive and negative examples. Working on positive examples only requires anti-unification and it allows solving the *characterisation task* (also called discovery or description problem), useful in KDD and Data-Mining. Working on positive and negative examples allows to perform the *discrimination task* (also called prediction problem or concept learning) that is the usual in most ML applications.

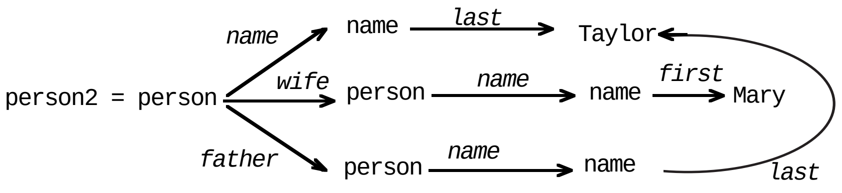
In the next sections we define some concepts common to the inductive learning methods we define in following chapters, i.e. the anti-unification operation, the definition of both discrimination and characterisation task, and how domain knowledge (usually called background knowledge in ILP) can be represented in the NOOS feature term formalism.

1.1. The Anti-unification Operation

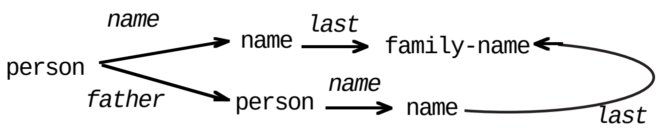
Intuitively, the anti-unification of two feature terms gives *what* is common to both feature terms (yielding the notion of generalisation) and *all* that is common to both (the most specific generalisation). The anti-unification is applied over feature terms in normal form i.e. the features of the feature terms have to be a value which can be obtained by evaluating either a closed method or a path reference (see section 3.2.1 in chapter 3).

Let *person1* and *person2* be the objects represented as the following NOOS feature terms





the anti-unification of both, is the following feature term:



i.e. the anti-unification feature term belongs to the sort *person* and has as features the common features to both *person1* and *person2* (`name` and `father`), The `last` feature of the *name* feature term appears in *person* with the value *family-name* that is the greatest lower bound in the sort hierarchy according to the ≤ sort relation, i.e. the most specific sort common to both *Taylor* and *Smith* sorts. Conversely, the features `wife` and `lives-at` only appear in one of the objects, so they do not appear in the anti-unification feature term. Formally, when a feature does not appear in a feature term is equivalent to consider that this feature has value *any*, the more general sort according to the ≤ sort relation. In such situation, the anti-unification of *any* with other value produces as result *any*, thus the feature will not appear in the anti-unified feature term. Notice that the path equality `person.name.last = person.father.name.last` of feature terms *person1* and *person2* also appears in the anti-unification feature term *person*. In general, a feature term obtained by the anti-unification of a set of feature terms will contain a path equality only if all the anti-unified feature terms contain the same path equality.

Formally, the anti-unification of a set of feature terms yields a greatest lower bound with respect to subsumption ordering. Thus, the *anti-unification* (AU) in feature terms is defined in the classical way (as the *least common subsumer* or *most specific generalisation*) over the subsumption lattice as follows:

- The *anti-unification* of two feature terms in normal form $\psi \sqcap \psi'$ is an greatest lower bound with respect to the subsumption ($\sqsubseteq$) ordering.

The anti-unification concept was introduced in the NOOS language as result of our work in inductive methods using the feature terms formalism (Armengol and Plaza, 1997).

```
Let D be a term with Sort (D) = Sort($E_1$)⊓Sort($E_2$)
Function AU2 ($E_1$, $E_2$, D)
   A = {$A_i$ | common attributes to $E_1$ and $E_2$}
   for each $A_i$ ∈ A do
      $V_i$ = ($v_{i1}$, $v_{i2}$) where $v_{i1}$ = $E_1$.$A_i$ and $v_{i2}$ = $E_2$.$A_i$
      if $v_{i1}$ = $v_{i2}$
         then add-feature(D, $A_i$, $v_{i1}$)
         else if there is a path p = ($V_j$, $d_j$) ∈ *paths* such that $V_i$= $V_j$
                then add-feature(D, $A_i$, $d_j$)
                else Let $d_i$ be a new term with sort Sort($d_i$)=Sort($v_{i1}$)⊓Sort($v_{i2}$)
                     add ($V_i$, $d_i$) to *paths*
                     add-feature(D, $A_i$, AU2($v_{i1}$, $v_{i2}$, $d_i$))
              endif
      end -if
   end-for
   return D
end function
```

Figure 4.3. Anti-unification operation that constructs the most specific generalisation covering a given set of positive examples. The function *Add-feature(d, a, v)* adds the feature *a* with value *v* to the description *d*.

In our framework, examples useful for induction are contained in the `solved episodes` models. Each model contains a feature term that is the description of an already solved problem and the solution obtained for it. In particular, the anti-unification operation uses a set of `solved episodes` models containing positive examples (we call E^+ to this set) to construct the most specific generalisation D subsuming all the examples in E^+. As we will see later, when feature terms have sets of values in some feature, anti-unification may be not unique. Figure 4.3 shows the algorithm AU2 used to obtain a most specific generalisation of two examples E_1 and E_2.

The anti-unification, AU2 (E_1, E_2, D), is applied to two examples E_1 and E_2 (represented as feature terms in normal form) and produces a feature term D containing the features that are common to both E_1 and E_2. The values of the features in D have to satisfy the following conditions:

1) If a feature *f* has the same value *v* in both examples E_1 and E_2, the value of *f* in D is *v*.

2) If a feature *f* has value of sort s_1 in E_1 and value of sort s_2 in E_2, the value of *f* in D is the most specific sort common to s_1 and s_2, i.e. the greatest lower bound of s_1 and s_2 in the $\leq$ sort order.

3) Otherwise, the examples E_1 and E_2 cannot be anti-unified.

Feature terms can also contain path equality, i.e. two features of a feature term having the same value. If two feature terms to be anti-unified have path equality between the same features, then the feature term resulting from the anti-unification also contains the same path equality (see previous example). Otherwise, path equality does not hold for the anti-unification feature term. For exposition convenience, we first explain AU2 assuming that the features have only one value. Later we will explain the case when feature terms have sets of values as value of some features.

Let us suppose that we want to anti-unify the feature terms E_1 and E_2. For each common feature A_i of E_1 and E_2, let us consider the pair $V_i = (v_{i1}, v_{i2})$, where $v_{i1} = E_1.A_i$ (the value or sort taken by the A_i feature in the example E_1) and $v_{i2} = E_{i2}.A_i$. The first step in the anti-unification of the pair $V_i = (v_{i1}, v_{i2})$ is to search for path equality. Path equality means that the same pair of values has already appeared in another feature. In the implementation, we use the variable called **paths** (see algorithm in figure 4.3) that contains all the pairs (V_j, d_j) already processed and where d_i in AU2(v_{j1}, v_{j2}, d_i) is the feature term generated by anti-unifying the values of the pair $V_j = (v_{j1}, v_{j2})$. Therefore, given the pair V_i the algorithm searches in **paths** for a pair (V_j, d_j) such that $V_j = V_i$. If this pair is found it means that there is path equality because the pair V_i has already been encountered by the AU2 algorithm. In order to also have the path equality in the anti-unified term, the value for A_i has to be exactly d_j. When there is no pair V_j in **paths** such that $V_j = V_i$ the anti-unification of the values v_{i1} and v_{i2} has to be computed according to the three conditions above.

Feature terms in NOOS, as we saw, are set-valued. Let us suppose that $S_1 = E_1.A_i$ and $S_2 = E_2.A_i$ are sets of values. Intuitively, the anti-unification of S_1 and S_2 has to produce as result a set S. Each element in S is the anti-unification of a value of S_1 and a value of S_2. There are $N=\text{Card}(S_1) \times \text{Card}(S_2)$ possible combinations of pairs of values from S_1 and S_2. However, the set S will contain min $\{\text{Card}(S_1),\text{Card}(S_2)\}$ more specific combinations.

Specifically, the anti-unification of S_1 and S_2 finds a set of values S such that:

1) Card (S) = min $\{\text{Card}(S_1), \text{Card}(S_2)\}$

2) each $s_i \in S$ is obtained from the anti-unification of two values $v_j \in S_1$ and $v_k \in S_2$. The AU2 algorithm is applied to each possible pair (v_j, v_k) where $v_j \in S_1$ and $v_k \in S_2$, obtaining a set $\{g_p\}$ containing $\text{Card}(S_1) \times \text{Card}(S_2)$ descriptions.

From the set $\{g_p\}$, a most specific combination[1] of Card(S) elements has to be taken as the value set S of the feature A_i. Two remarks can be made when the set S is constructed:

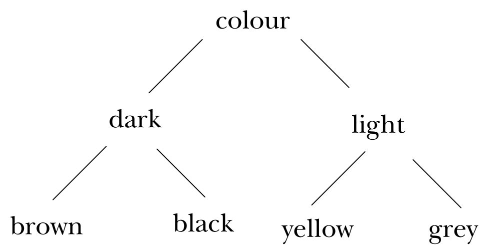
1. $\forall g_i$ from AU2(v_i, v'_i, g_i) such that $g_i \in S$ and $\forall g_j$ from AU2(v_j, v'_j, g_j) such that $g_j \in S$, where $v_i, v_j \in S_1$ and $v'_i, v'_j \in S_2$ it is satisfied that $v_i \neq v_j$ and $v'_i \neq v'_j$.
2. Several incomparable combinations that are maximally specific can exist. AU2 randomly chooses one of them.

As an example, let us suppose that objects OBJ1 and OBJ2 have the following definitions:

[1] A combination is most specific if it is not subsumed by any other combination.

```
(define (object OBJ1)              (define (object OBJ2)
   (COLOURS yellow brown))            (COLOURS grey black))
```

where the values *yellow, brown, grey* and *black* belong to the following sort hierarchy:



The result of the anti-unification of OBJ1 and OBJ2 is an object with a `colours` feature has two values. These two values are obtained from the anti-unification of the values that OBJ1 and OBJ2 have in the `colours` feature. The table 4.1 shows the different pairs (c1 ... c4) composed of one value of OBJ1 and one value of OBJ2. The column labeled as *glb* contains the greatest lower bound of the sorts of the two values of the associated pair. For instance, with respect to the sort hierarchy above, the greatest lower bound of *yellow* and *grey* is *ligth* since this is the most specific sort common to both values, while the greatest lower bound of yellow and black is colour.

	OBJ1	OBJ2	glb
c1	yellow	grey	ligth
c2	yellow	black	colour
c3	brown	grey	colour
c4	brown	black	dark

Table 4.1. Combinations of values from the `colours` feature in objects OBJ1 and OBJ2 and their respective greatest lower bound (lub).

So the object from the anti-unification of OBJ1 and OBJ2 could contain in its `colours` feature any of the following pairs: (c1, c2), (c1, c3), (c1, c4), (c2, c3), (c2, c4), (c3, c4).

We say that a pair of combinations are *compatible* when they have a different set of values. For instance, the pair (c1, c2) is not compatible since c1 has as elements *yellow* and *grey* (see table above) and c2 has as elements *yellow* and *black.* This situation also occurs in the pairs (c1, c3), (c2, c4), (c3, c4). In other words, the compatible pairs are (c1, c4) and (c2, c3). As a consequence, the following two expressions are candidates to be the anti-unification of OBJ1 and OBJ2:

```
(define (object OBJ3)              (define (object OBJ4)
   (COLOURS ligth dark))             (COLOURS colour colour))
```

OBJ3 is obtained taking the *lub* values from c1 and c4 of the table above and OBJ4 is obtained taking the *lub* values from c2 and c3. Notice that OBJ4 subsumes OBJ3, i.e. OBJ3 is most specific than OBJ4. Because the anti-unification is defined as the least general generalization of two objects, the anti-unification of OBJ1 and OBJ2 is the object OBJ3.

1.2. Background Knowledge

Background knowledge is usually represented in ILP systems as rules representing relations among several concepts. Using NOOS feature terms, the background knowledge can be expressed by means of sorts, feature path references, and methods.

An example of domain where the background knowledge is necessary is the Traffic Law dataset used by the MOBAL system (Morik et al., 1993), where background knowledge is modelled as Prolog-like rules. Some inferential relations are the following:

$$\text{sidewalk (X)} \rightarrow \text{no-parking (X)}$$
$$\text{bus-lane (X)} \rightarrow \text{no-parking (X)}$$
$$\text{involved-vehicle (X,Y)} \wedge \text{major-corrosion (Y)} \rightarrow \text{unsafe-vehicle-violation(X)}$$
$$\text{involved-vehicle (X,Y)} \wedge \text{faulty-brakes (Y)} \rightarrow \text{unsafe-vehicle-violation(X)}$$
$$\text{involved-vehicle (X,Y)} \wedge \text{worn-tires (Y)} \rightarrow \text{unsafe-vehicle-violation(X)}$$

Some background knowledge using feature terms is expressed using sorts while other inferential relations are expressed as paths referencing the value of a feature. Thus, the relations above are expressed as feature terms as follows:

```
(define place)
(define (place no-parking-place)
        (no-parking true))
(define (no-parking-place Sidewalk))
(define (no-parking-place Bus-lane))
(define (place parking-place)
        (no-parking false))
(define (parking-place road-edge))

(define event
   (involved-vehicle )
   (parking-violation (>> no-parking car-parked involved-vehicle))
   ((unsafe-vehicle-violation (reify (>> major-corrosion involved-vehicle))
                              (reify (>> faulty-brakes involved-vehicle))
                              (reify (>> worn-tires involved-vehicle)))))
```

In other words, we have defined a sort called *place* and a refinement (subsort) of it called *no-parking-place*. Objects belonging to the *no-parking-place* sort have a feature called `no-parking` with value *true*. *Sidewalk* and *bus-lane* are subsorts of *no-parking-place*, consequently they also have the feature `no-parking` with value *true*.

Notice that in the definition above of the *event* concept the `parking-violation` feature has the following path reference to compute its value:

```
        (>> no-parking car-parked involved-vehicle)
```

This reference is evaluated before the application of learning methods, since feature terms are used in normal form.

The three rules used in MOBAL to obtain a value for `unsafe-vehicle-violation` are represented by the three paths in the `unsafe-vehicle-violation` feature in the definition of *event*. Thus, the value of the `unsafe-vehicle-violation` feature can be computed using one of the three following path references: 1) the value of the `major-corrosion` feature of the involved vehicle, 2) the value of the `faulty-brakes` feature, and 3) the value of the `worn-tires` feature. If any of these path references returns *true* as result, the value of the `unsafe-vehicle-violation` feature will be also *true*, otherwise its value is *fail*.

No closed methods appear in this example i.e. the possibility of describing a feature value by means of a method (see section 3.2.1 in chapter 3).

In chapters 5 and 6 there are the results of the two inductive learning methods INDIE and DISC over the Traffic Law dataset.

1.3. The Discrimination Task

The process of induction over feature terms with the goal of finding a *discrimination description* can be specified as follows:

Given: a set of positive E^+ and negative E^- examples, a notion of subsumption and background knowledge in the form of domain methods

Find a feature term (description) ψ such that $\forall\ e \in E^+ : \psi \sqsubseteq e$ and $\forall\ e' \in E^- : \psi \not\sqsubseteq e'$

In other words, a discriminant description ψ subsumes all the positive examples and does not subsume negative examples. A discriminant description ψ represents the description of a concept in the context of other classes (or at least a negative class). This description is useful to predict if an unseen example belongs to the described concept in front of other classes (or at least a negative class).

1.4. The Characterisation Task

We define the process of induction over feature terms with the goal of finding a *characteristic description* as follows:

Given: a set of positive examples E^+, a notion of subsumption and background knowledge in the form of domain methods

Find a feature term (description) ψ such that $\forall\ e \in E^+ : \psi \sqsubseteq e$.

In other words, the characterisation task searches for a description ψ subsuming all the positive examples (without taking into account negative examples). The characteristic description ψ can be obtained by the anti-unification of the positive examples and represents all the regularities common to all of them. Thus, ψ, (namely the least general generalisation of E^+) subsumes all positive examples (but it may also subsume some negative example if they existed).

2. Lazy Learning Methods using Feature Terms

As we have explained in section 3 of chapter 2, learning methods can be classified as eager and lazy methods. In section 1 of current chapter we already defined inductive methods, that are the most typical eager methods. As we said, inductive learning methods build general knowledge from specific knowledge.

Instead, lazy learning methods can be characterised as directly using past experience (typically a specific past example is used) to solve a current problem. The problem solving process for lazy learning can be viewed in our framework as a process of construction of a new `solved episode` model of the current problem from one (or more) `solved episode` model(s) of past example(s). In fact, the process of learning and the associated process of using the learned knowledge for solving new problems are now linked and cannot be analysed separately, as we could in purely inductive methods. We can also characterise lazy learning methods as those that learn only the extensional concept descriptions and do not generate (permanent) intensional descriptions. Thus, in lazy learning methods, past experience is used[2] in a problem-centred way to directly construct the new `solved episode` model of the current problem to solve.

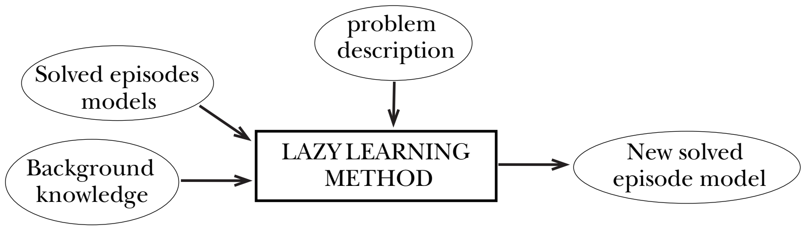


Figure 4.4. Representation of a lazy learning method.

Typical lazy methods are case-based reasoning methods that retrieve past

[2] Essentially, using past experience to affect future outcomes is the hallmark of learning.

experiences, considered as relevant, and from them solve a current problem. In lazy learning methods, generalisation is produced on-demand and thus it is not included in a separate learning phase but it is incorporated inside the problem solving process (see figure 4.4). For instance, in CBR methods when trying to solve a new problem, the solution of an old problem (i.e. the `solved episode` model) is transferred (and possibly transformed) to the current problem (i.e. a new `solved episode` model). The `solved episode` model constructed during the problem solving process can be later used by the lazy learning method as experience amenable to be useful in solving future problems.

Another important issue in lazy learning methods is the use of *similar* precedents to solve a new problem. Approaches using estimations of similarity usually deal with attribute-value representations of cases. In these approaches similarity among cases is estimated using metrics. Structured representations of cases are more powerful but establishing similarity among them is still an open research issue (Börner, 1993; Bunke and Messmer, 1994). The proposed approaches use the notion of structural similarity in different ways having a common basic intuition: the similarity between two structured cases to be captured is about the structural relations that are common to both cases.

Plaza (1995) proposes a symbolic approach to estimate the similitude among cases. Given a new problem to solve p and a base of correctly classified precedents $E = \{e_i\}$, the main idea of this approach is to build a feature term S_{pe_i} (obtained by anti-unifying p with a precedent e_i) called *similitude term*, containing the commonalties among p and a precedent e_i. There is a similitude term relating the new problem p with each precedent $e_i \in E$. This similitude terms can be partially ordered taking benefit of the subsumption relation among feature terms. So, if $S_{pe_i} \sqsubseteq S_{pe_j}$ this means that e_j is more similar to p than e_j (that is p and e_j have more features in common).

In (Plaza et al., 1996) we have completed this approach by designing the Case Retrieval and Assessment using Symbolic Similitudes (CRASS) method. CRASS follows the idea of Plaza in building the similitude terms. The improvement with respect to the previous approach is that similitude terms are ordered according to an entropy measure.

In chapter 7 we propose a lazy learning method, LID, based on these ideas (similitude terms and subsumption) and also on a heuristic of entropy reduction. LID improves the discrimination power of previous approaches since it uses domain knowledge, such as the partition of the precedents in classes.

In the next chapters we describe two inductive learning methods, INDIE and DISC, and in chapter 7 we describe a lazy learning method, LID. INDIE is a heuristic bottom-up inductive learning method that uses the anti-unification operation and the subsumption relation to build a most specific generalisation describing a class. DISC is a heuristic top-down inductive learning method that builds a most general discriminant description for a class using anti-unification and subsumption. Finally, LID is a lazy learning method that, in a problem-centred way and using an entropy-reduction heuristic, builds a description capable to classify a current problem.

Chapter 5

The INDIE Method.

1. Introduction

The method proposed in this chapter, INDIE, is a heuristic bottom-up inductive learning method based on the subsumption relation among feature terms. INDIE can be used to solve both the characterisation task and the discrimination task. To solve the *characterisation* task (also called *discovery* or *description problem*) INDIE only needs positive examples. This task is useful in Knowledge Discovery in Databases and in Data Mining. To solve the *discrimination* task (also called *prediction problem* or *concept learning*) INDIE requires positive and negative examples. The discrimination task is the usual in most ML applications. The main contributions of INDIE are 1) the possibility of handling objects represented as feature terms and 2) the INDIE's soundness in handling imperfect data (i.e. data with noise and/or having unknown values).

In the following section we provide a general view of INDIE by means of a knowledge modelling analysis of the discrimination task. We also detail how imperfect data are handled in INDIE. In section 3 we provide a detailed explanation of the INDIE algorithm. Finally, in section 4 we provide results of the INDIE evaluation.

2. General View of INDIE

In this section we describe the INDIE method in solving the discrimination task following the framework described in chapter 2. In section 1.3 of chapter 4, we have defined *discrimination description* as follows:

Given: a class C, a set E^+ containing the positive examples of C, a set E^- containing negative examples of C, a notion of subsumption ($\sqsubseteq$) and background knowledge given in the form of domain methods

Find a description D for the class C such that $\forall$ e $\in$ C : D $\sqsubseteq$ e and $\forall$ e' $\in$ C: D $\not\sqsubseteq$ e'

In other words, a discrimination description D for a class C subsumes all the positive examples of C and does not subsume the negative examples of C.

Let us suppose that training examples E are classified in N solution classes $C_1,\ldots, C_N$. The goal of INDIE in the discrimination task is to build a description D_k for a solution class C_k such that D_k subsumes all the positive examples and does not subsume negative examples of C_k. Positive examples of the solution class C_k are those training examples classified as belonging to C_k. Negative examples of C_k are those belonging to solution classes different than C_k. We assume that the training examples are correctly classified.

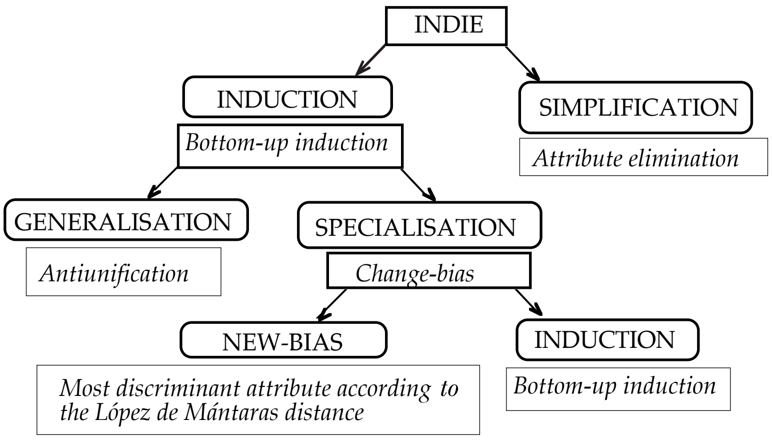


Figure 5.1. Task/method decomposition of the INDIE method for the discrimination task.

Figure 5.1 shows the knowledge modelling of INDIE in the discrimination task. Thus, INDIE decomposes in two subtasks: `induction` and `simplification`. The `induction` task obtains general knowledge from positive and negative examples, obtaining a description for one solution class. The `simplification` task generalises as much as possible the description obtained by `induction` task. The main goal of `simplification` task is the elimination of non-relevant features using a post-process similar to the one used by FOIL (Quinlan, 1990).

The `induction` task is solved using the `Bottom-up-Induction` method. This is the heuristic bottom-up method that builds a description that subsumes all the positive examples and does not subsumes negative examples. This method decomposes in two subtasks: `Generalisation` and `Specialisation`. The `Generalisation` task uses the `anti-unification` method explained in section 1.1 of chapter 4 to obtain a most specific generalisation D subsuming all the positive examples.

If the description D obtained by `Generalisation` task also subsumes some negative examples, the `Specialisation` task has to be used to specialise D. The specialisation is made by means of the `Change-bias` method that replaces D with a disjunction of descriptions. This method decomposes in two subtasks: `new-bias` and `induction` . The `new-bias` task decides how many descriptions are necessary using López de Mántaras distance to obtain a partition of the training examples. Finally, the `induction` task is newly invoked for each partition set. This process is repeated until a description that does not subsume negative examples is found.

3. Description of the INDIE Algorithm

Given a set of training examples E = $\{e_1,\ldots,e_m\}$ and a set of solution classes C= $\{C_1,\ldots,C_n\}$, the goal of INDIE is to obtain a discrimination description D_k for each solution class C_k. Each example $e_i \in$ E is a feature term having a subset of features $A_i = \{A_{i1},\ldots, A_{in} \mid A_{ij} \in F\}$, where F is the set of features appearing in any domain object (example).

```
D = ø
Function INDIE (E+, E-)
  Dk = Anti-unification(E+) ; most specific generalisation
  if there is some e ∈ E- such that Dk ⊑ e
       then Al = {Ai | features in Dk chosen according to a bias}
            PN = Discriminant-partition (Al,E+,E-)
                for each set Si ∈ PN do
                        Di = INDIE(Si, E-)
                        Add Di to D
                end for
       else Add Dk to D
  end-if
  Eliminate any d ∈ D such that d ⊑ d' ∈ D
  return D
end-function
```

Figure 5.2. INDIE obtains a set of descriptions D that do not subsume negative examples for the current class.

A description $D_k = \{d^j{}_k\}$ represents a disjunction of feature term descriptions for the current solution class C_k. Each $d^j{}_k$ subsumes a subset of positive examples of C_k and does not subsume negative examples. In a discrimination task, negative examples of a solution class C_k are all those training examples that do not belong to C_k.

Given a set of positive examples E^+ for a solution class C_k the INDIE algorithm (figure 5.2) obtains, using the anti-unification operation, a most specific generalisation D_k subsuming all the examples in E^+. If the description D_k does not subsume negative examples then D_k is a correct description for C_k. Because the value of a feature can be a set, several most specific generalisations subsuming all the positive examples can be built. This means that if one of the most specific generalisations D_k subsumes some negative example there are two options to solve this situation:

1) to search for another most specific generalisation D_j that does not subsume negative examples, or

2) to specialise D_k until no negative examples are subsumed.

Using the first option, all the possible most specific generalisations are tested searching for a description D_k that does not subsume negative examples. If all the descriptions D_k subsume negative examples, the second option has to be taken. In other words, the first option warrants the completeness of INDIE. The second option, assumes that the only way to describe the current solution class C_k is using a disjunction of descriptions. In the current implementation of INDIE we have taken the second option.

The next question is, how many descriptions are necessary to describe C_k? To answer this question a heuristic approach is taken. INDIE selects the most relevant feature A_d (using the *discriminant-partition* function explained in section 3.3). Each feature $f \in A_i$ induces several partitions over the set of training examples, according to values and sorts that the feature f can take (see section 3.3 for a more detailed explanation). The most discriminant feature A_d is a feature inducing a partition P_N of the training examples such that the López de Mántaras distance between P_N and the correct partition is minimum (see section 3.3). Thus, the induced partition P_N has N classes where N is the number of different values or sorts that A_d takes in E^+. Therefore, if D_k subsumes some negative example then INDIE will generate a disjunct of (at most) N descriptions for C_k. This new specialised description is $D_k = \{d^j_k\}$, for j = 1 to N, obtained by applying the INDIE algorithm to each set of the partition P_N.

This process is repeated until D_k does not subsume negative examples or all the features have been used. Finally, if there are two descriptions d^1_k and d^2_k in D_k such that $d^1_k \leq d^2_k$ then d^2_k can be eliminated and the disjunct is simplified.

In the next sections, we explain the bias used to select the set of features allowing the definition of a partition of the positive examples and how the most discriminant partition is selected (*discriminant-partition* operation).

3.2. Bias

Let us suppose that D_k is a description obtained from the anti-unification of a (sub)set of positive examples E^+, and that D_k subsumes some negative examples. INDIE specialises D_k selecting a feature A_d that induces a partition P over E^+ (see in next section how this partition is induced). In this section we explain which is the bias used to determine the set of features candidates to specialise D_k.

We could consider as candidates to produce this partition any feature used to describe a training example. Nevertheless, when there are imperfect data (unknown values) training examples can be described using a different set of features. Thus, a possible bias may be to take only those features appearing in D_k (the last description obtained by anti-unification). This bias reduces the set of candidates to those features that all the positive examples have in common. In a structured representation two kinds of features can be distinguished: leaf features and intermediate features (those belonging to intermediate levels of the structured representation).

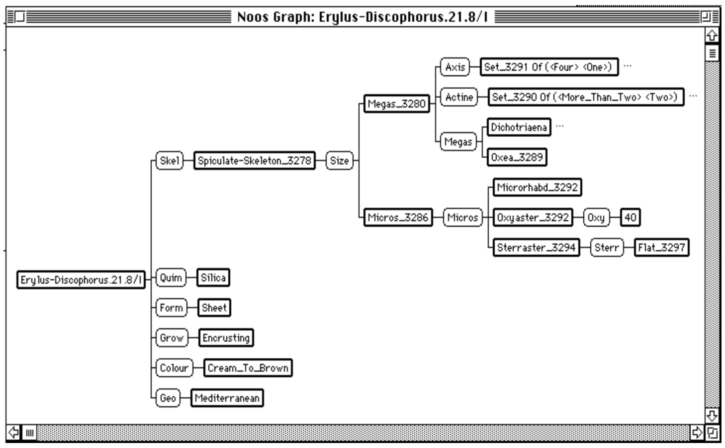


Figure 5.3. Description of a feature term representing a marine sponge.

For example, in figure 5.3 some leaf features are `axis` or `grow` and some intermediate features are `size` or `micros`. Usually, existing inductive learning methods select one predicate at a time to specialise a clause. If we select an intermediate feature as the most discriminant, its value is a subterm and this means that the description D_k is specialised according to the sorts in that subterm. This situation is equivalent to specialise a clause introducing several predicates at a time. The selection of only one predicate is achieved in feature terms by selecting a leaf feature. Therefore, our bias is to consider as candidates the set of leaf features of D_k. In practice, the selection of an intermediate feature to specialise the description D_k tends to produce descriptions too specific (in the sense that it quickly reaches a disjunction of M descriptions, where M is the number of positive examples). In principle, we are interested in a description of a solution class using the least number of descriptions.

Let us suppose now the following description of a feature term:

$$\text{Francesca : female} \left[\begin{array}{l} \text{husband} = \text{Piero} \left[\begin{array}{l} \text{wife} = \text{Francesca} \\ \text{hair} = \text{blond} \end{array}\right] \\ \text{son} = \text{Marco} \left[\begin{array}{l} \text{mother} = \text{Francesca} \\ \text{father} = \text{Piero} \\ \text{hair} = \text{brown} \end{array}\right] \\ \text{hair} = \text{brown} \end{array}\right]$$

Which are the leaf features of the *Francesca* feature term? This is not a question easy to answer. We can define a *leaf feature* as a feature whose value is a feature term without features. Features as `axis` or `grow` in figure 5.3 satisfy this definition. Nevertheless, according to this definition of leaf feature only the `hair` feature of *Francesca* is a leaf, since the values (feature terms) of the remaining features (i.e. `husband`, `son`) have features. To deal with this kind of feature terms we introduce the notion of depth of a feature:

- The *depth of a feature* F in a feature term X is the number of features that compose the path from the root of X to F.

According to this definition, given a depth N we can define

- The *leaf features* of a feature term are the set of features whose path from the root has length N.

For instance, the features `hair`, `husband` and `son` of the feature term *Francesca* have depth 1, whereas the features `hair`, `mother`, `father` and `wife` from *Piero* and *Marco* have depth 2. INDIE allows the construction of feature terms (descriptions) having a predetermined depth (or minor if the feature term value has not features) as a user-given parameter. Thus, INDIE will consider as leaves of a feature term those features in a pre-determined depth. For example, let us suppose that we want to consider as leaves those features having depth equal than 3. In such situation, the term *Francesca* having depth 3 is the following:

$$\text{Francesca : female} \left[\begin{array}{l} \text{husband} = \text{Piero} \left[\begin{array}{l} \text{wife} = \text{Francesca} \left[\text{husband} = \text{Piero}\right] \\ \text{hair} = \text{blond} \end{array}\right] \\ \text{son} = \text{Marco} \left[\begin{array}{l} \text{mother} = \text{Francesca} \left[\begin{array}{l} \text{husband} = \text{Piero} \\ \text{son} = \text{Marco} \\ \text{hair} = \text{brown} \end{array}\right] \\ \text{father} = \text{Piero} \\ \text{hair} = \text{brown} \end{array}\right] \\ \text{hair} = \text{brown} \end{array}\right]$$

where the leaves are `hair`, `husband` (from *Piero*), `hair`, `husband` and `son` (from *Francesca* mother of *Marco*), `hair` and `father` (from *Marco*) and `hair` (from *Francesca*).

In section 4.1.7 we explain the results of INDIE over the Mesh dataset. In that domain the feature term depth is controlled in order to induce descriptions of the solution classes.

3.3. Partitioning the Set of Positive Examples

In this section we explain how is induced a partition in the set of training examples $E = E^+ \cup E^-$. Figure 5.4 shows the algorithm followed to induce this partition.

```
Function DISCRIMINANT-PARTITION ($A_l$, $E^+$, $E^-$)
   Dist = ∅
   while $A_l \neq \emptyset$ do
      $P_c$ = (($E^+$)($E^-$))  ;; the correct partition
      for $A_i \in A_l$ do
         $P_i = \{S_i \subset E \mid \forall v_i \in S_i$ and $\forall v_j \in S_j$: Sort($v_i$)$\neq$Sort($v_j$)$\}$
         $D_i$ = D($P_c$, $P_i$)  ;; López de Mántaras distance
         Add $D_i$ to Dist
      end-for
   end-while
   while Dist $\neq \emptyset$ and (useful-feature = false) do
      $d_{min}$ = min $\{D_i \in$ Dist$\}$
      Let $A_{min}$ and $P_{min}$ be the feature and the partition associated to $d_{min}$
      $P_d = \{S'_i \mid S'_i = S_i - E^-: S_i \in P_{min}\}$
      if $P_d$ has only one non-empty $S'_i$
              then remove $d_{min}$ from Dist
              else  useful-feature = true
      endif
   end-while
   if Dist = ∅ then return $E^+$
               else return $P_d$
   end-if
end-function
```

Figure 5.4. Discriminant-partition function selects the most useful feature A_{min} in a description using the López de Mántaras distance. The outcome is the partition induced by A_{min}. If there is not a most useful feature, Discriminant-partition returns the set of positive examples E^+.

Let D be the description obtained from the anti-unification of the set of positive examples E^+. When D subsumes negative examples we want to induce a partition of the set E^+ in subsets $E_1 \dots E_n$ to which the anti-unification will be recursively applied. Let $A_l = \{A_1 \dots A_n\}$ be the set of features that can be used to induce a partition in E^+. The selection of a feature $A_d \in A_l$ is made using the López de Mántaras distance (1991). The main idea is that each feature $A_i \in A_l$ induces a partition P_i over the training set E according to the values and sorts that A_i can take in E.

Usually, the partition P_j induced by a feature A_j is built according to the number of different values that A_i takes in the training set. Thus, examples belonging to a set S_i of the partition P_j (i.e. $S_i \subset P_j$) have the same value in the feature A_j. INDIE obtains this partition according to the values of the feature A_j, but also generates other partitions taking into account the sort hierarchies to which those values pertain. In other words, for each feature A_j more than one partition $\{P_{jk}\}$ can be generated. Each partition P_{jk} is induced according to different combinations of the sorts to which the possible values of A_j belong.

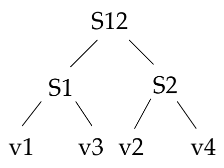


Figure 5.5. An example of sort hierarchy.

Let us suppose that a feature A_j takes in the training set the values v_1, v_2 and v_3 that are refinements of the sorts S_1, S_2, and S_{12} according to the hierarchy of sorts in figure 5.5 (notice that v4 is not considered). In addition to the partition induced by v_1, v_2 and v_3, INDIE generates the following partitions:

- (S1, v_2), i.e. the training set is divided in two subsets, one containing examples whose value in the feature A_j belongs to the sort S_1 and the other containing examples whose value in the feature A_j is v_2.
- (v_1, v_3, S_2), i.e. the training set is divided in three subsets, one containing examples whose value in the feature A_j is v_1; a second subset containing examples whose value in the feature A_j is v_3 and finally a third subset of examples whose value in the feature A_j is of sort S_2.
- (S_1, S_2), i.e. the training set is divided in two subsets. One subset contains examples whose value in A_j is of sort S_1. Examples in the other subset are those whose value in the feature A_j is of sort S_2.

Notice that in the two last partitions v_2 is generalized by the sort S_2, and the sort S_2 also includes the value v_4 (not considered in the training set). This means that the description obtained from these partitions could subsume new examples having the feature A_j taking the value v_4.

Thus, for each $A_j \in A_l$, INDIE induces partitions by taking into account the sort hierarchy and computes their López de Mántaras distance (LMD) distance to the correct partition. The LMD measures the distance between P_i and the correct partition[1]. Given two partitions P_A and P_B of a set S, the LMD between them is computed as follows:

[1] In INDIE the correct partition has two sets, one containing the positive examples and the other containing the negative examples.

$$D_N(P_A, P_B) = 2 - \frac{I(P_A) + I(P_B)}{I(P_B \cap P_A)}$$

where

$$I(P_A) = -\sum_{i=1}^{n} P_i \log_2 P_i, \quad P_i = \frac{|X \cap C_i|}{|X|} \text{ and } I(P_A \cap P_B) = -\sum_{i=1}^{n}\sum_{j=1}^{m} P_{ij} \log_2 P_{ij}$$

P_i is the probability of occurrence of each class C_i in the set of examples X, i.e. the proportion of examples in X that belong to C_i. $I(P_A)$ measures the information contained in the partition P_A. $I(P_A \cap P_B)$ is the mutual information of two partitions.

The LMD over feature terms provides the following relation among features:

- Let P_c be the correct partition, P_j and P_k the partitions induced by features A_j and A_k, we say that feature A_j is *more discriminant than* feature A_k iff $LMD(P_c, P_j) \le LMD(P_c, P_k)$

Then, the feature A_d having a minimum LMD measure is selected as the more discriminant. Using the more discriminant feature $A_d \in A_l$ inducing a partition in the set of positive examples in subsets $E_1 \dots E_n$, two situations can be found:

1) There is a set E_i containing all the positive examples (and perhaps some negative examples) and the remaining sets E_j only contain negative examples.

2) The positive examples are distributed in several sets E_i.

The anti-unification has to be applied to positive examples only. Thus, in the first situation the negative examples are removed from the sets $E_1 \dots E_n$; it may happen that only one set E_i remains that, in fact, is E^+. Therefore, the anti-unification would produce the same description D that is the current hypothesis in INDIE. In this situation, the feature A_d is removed from A_l and another feature $A_i \in A_l$ having a second best LMD is chosen. This process is repeated until the second situation is produced.

In the second situation we obtain a partition P'_i containing only positive examples, thus the INDIE algorithm is applied to each set E_i in P'_i. Notice that the first time that INDIE is used, the training set contains all the positive and negative examples. When, INDIE is recursively applied the training set contains a subset of the positive examples and all the negative examples.

3.4. Generalisation post-process

After applying INDIE, an optional post-processing step can be used. Since $D_k = \{d^j{}_k\}$ is a most specific generalisation for a solution class C_k, it can be generalised (in principle) as far as no negative example is subsumed (see figure 5.6). For each description $d^j{}_k \in D_k$, the algorithm for post-processing uses López de Mántaras distance to rank all the features belonging to d^j_k. The features are considered from the least discriminant to the most discriminant. Following this order, each step in the algorithm considers a new general description generated by eliminating the least discriminating feature from a description d^j_k. If the new description does not subsume negative examples, then the least discriminant feature can be eliminated. All the features are explored in this order, and the final result is a description containing the features that are necessary to discriminate the examples of the current class C_k. The resulting description is one of the most general discrimination descriptions that describe the current solution class C_k.

```
Function Generalised-INDIE ($E^-$, $D_k$)
  $D_k = \{d^i_k$ | disjunctive description for the solution class $C_k\}$
  For each description $d^i_k \in D_k$ do
    $A_o = (A_1 \ldots A_n)$;Features in $d^i_k$ ordered using the LMD heuristic
    For each $A_i \in A_o$ (i = 1 to n) do
         $d_{new} := d - A_i$
         if there is no $e \in E^-$ such that $d_{new} \leq e$ then
            $d := d_{new}$
         end-if
    end-for
end-for
```

Figure 5.6. Generalised-INDIE algorithm that eliminates features from a current description obtained using the INDIE method.

4. Evaluation of the INDIE Method

Two aspects of the INDIE method have been evaluated: the suitability of inductively created descriptions and its predictivity degree. Suitability of predictions has been evaluated over several relational domains. Predictivity has been evaluated over Lymphography and Soybean databases which have been used to evaluate most of propositional learners in the literature. In the next sections both aspects are explained. The description of domains used to evaluate INDIE can be found in appendix A. In chapter 9 we analyse the INDIE behaviour when is applied to classification of marine sponges, where domain objects (sponges) are represented as feature terms.

4.1. Evaluation of the Descriptions Suitability

In this section we analyse the INDIE behaviour used as concept learner in relational domains. For the sake of readability, we analyse INDIE results using the simplification post-process, since it produces more compact descriptions.

Concept learning evaluation of INDIE has been made using domains (such as trains, families, etc) typically used in ILP and relational learning systems. In next sections we compare descriptions obtained by

INDIE with those obtained by ILP systems (FOIL, LINUS, KLUSTER,...) in the same domains.

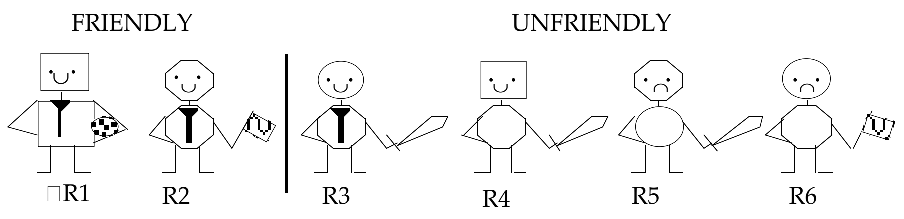


Figure 5.7. Robots used as input in the Robots dataset.

4.1.1. Robots Dataset

The Robots dataset (Lavrac and Dzeroski, 1994) consists of a description of six robots that belong to two solution classes: *friendly* and *unfriendly* (see figure 5.7). Each robot is described using an attribute-value representation with five features: `smiling`, `holding`, `has-tie`, `body-shape` and `head-shape`. In the follow we will describe step by step how INDIE obtains the description for the *friendly* and *unfriendly* class.

Let us consider the *friendly* class. The examples of this class (see figure 5.7) are R1 and R2 and the negative examples are R3, R4, R5 and R6. First, INDIE obtains the anti-unification of the examples. The result of this step is the following description:

$$D = X : \text{robot} \begin{bmatrix} \text{smiling} \doteq \text{yes} \\ \text{holding} \doteq Z : \text{object} \\ \text{has - tie} \doteq \text{yes} \\ \text{body - shape} \doteq Y : \text{shape} \\ \text{head - shape} \doteq Y : \text{shape} \end{bmatrix}$$

Notice that variable Y in description D is shared by two features (`body-shape` and `head-shape`), this means there is a path equality between both features. Due to this fact the description D does not subsumes negative examples, so it is a discriminate description for the *friendly* class. Using the generalisation post-process described in section 3.4 INDIE obtains the following relational definition for the *friendly* class:

$$\text{Friendly} = X : \text{robot} \begin{bmatrix} \text{body - shape} \doteq Y : \text{shape} \\ \text{head - shape} \doteq Y : \text{shape} \end{bmatrix}$$

According to this description, robots having the same variable Y in `body-shape` and `head-shape` (in other words, having the same body and head shape) belong to the *friendly* class. LINUS (Lavrac and Dzeroski, 1994) obtains the following rule as description of the *friendly* class:

Class = friendly **if** [smiling = yes] $\wedge$ [holding = balloon]
Class = friendly **if** [smiling = yes] $\wedge$ [holding = flag]

Lavrac and Dzeroski (1994) describe how background knowledge can be introduced in attribute-value learning. In addition to attributes describing domain objects, they suggest that the description of domain objects can include attributes representing relations between other attributes. These new attributes are like functions in the sense that their value is *true* or *false*. In particular, descriptions of robots can include a new attribute called `same-shape` that is *true* if both body and head have the same shape and *false* otherwise. According to this new description LINUS obtain for the *friendly* class the same description that INDIE does obtain directly.

Let us consider now how INDIE obtains a description for the *unfriendly* class. The examples of this class are R3, R4, R5 and R6 whereas R1 and R2 are considered as negative examples. First, INDIE anti-unifies the examples and obtains the following description:

$$D1 = X : \text{robot} \begin{bmatrix} \text{smiling} \doteq Z : \text{boolean} \\ \text{holding} \doteq R : \text{object} \\ \text{has-tie} \doteq P : \text{boolean} \\ \text{body-shape} \doteq N : \text{shape} \\ \text{head-shape} \doteq Y : \text{shape} \end{bmatrix}$$

However, this description subsumes all the negative examples, therefore INDIE has to specialise it. The first step of INDIE's specialisation takes each feature of D1 to induce a partition over the set of positive and negative examples. The following table shows the partitions induced by the features of D1:

Feature	Partition	Values
smiling	((R1 R2 R3 R4) (R5 R6))	(yes, no)
holding	((R1) (R2 R6) (R3 R4 R5))	(balloon, flag, sword)
has-tie	((R1 R2 R3) (R4 R5 R6))	(yes, no)
head-shape	((R1 R4) (R2 R5) (R3 R6))	(square, octagon, round)
body-shape	((R1) (R2 R3 R4 R6) (R5))	(square, octagon, round)

The second step of specialisation is to compute the López de Mántaras distance from each partition above to the correct partition P_c = ((R1 R2) (R3

R4 R5 R6)). INDIE uses the most discriminant feature to obtain a specialisation of the description D1. The feature inducing a partition with minimum López de Mántaras distance to the correct partition is the most discriminant. In our example, the most discriminant feature is `holding`. As a consequence, the partition P = ((R1) (R2 R6) (R3 R4 R5)) will be used to specialise D1.

In the third step of specialisation, INDIE removes the negative examples from the partition sets, therefore the partition P is transformed in the partition P' = ((R6) (R3 R4 R5)). The fourth step is to recursively apply INDIE to each set of the partition P'.

In particular, the anti-unification of the set (R6) is a description D3 that is equal to the description of R6. The anti-unification of the set (R3 R4 R5) is the description:

$$D2 = X : \text{robot} \begin{bmatrix} \text{smiling} & \doteq & Z : \text{boolean} \\ \text{holding} & \doteq & \text{sword} \\ \text{has-tie} & \doteq & P : \text{boolean} \\ \text{body-shape} & \doteq & N : \text{shape} \\ \text{head-shape} & \doteq & Y : \text{shape} \end{bmatrix}$$

This description does not subsume negative examples. Therefore, the description of the class *unfriendly* is the disjunction of the descriptions D1 and D2.

Optionally, the generalisation post-process can be applied separately to both disjuncts. The descriptions obtained by INDIE after the application of the generalisation post-process is a disjunction of two feature terms:

$$\begin{aligned} \text{Unfriendly} = {} & X_1 : \text{robot} \left[\text{has-tie} \doteq \text{no} \right] \\ & \vee \\ & X_2 : \text{robot} \left[\text{holding} \doteq \text{sword} \right] \end{aligned}$$

i.e. a robot is *unfriendly* when either it has not tie or whether it holds a sword. LINUS using any of learners that includes (NEWGEM, ASSISTANT or CN2) obtains the following rule:

Class = unfriendly **if** [smiling = no]
Class = unfriendly **if** [smiling = yes] $\wedge$ [holding = sword]

It is worth noting that the disjunct [`smiling` = no] is equivalent to the condition [`has-tie` = no] obtained by INDIE. During INDIE's post-process, features contained in the obtained description are ranked according to their relevance using López de Mántaras distance. For the *unfriendly* class the features `smiling` and `has-tie` have the same distance value and, INDIE randomly chooses one of them for elimination. In other words, INDIE could also obtain the following description:

$$\begin{aligned} \text{Unfriendly} = {} & X_1 : \text{robot} \left[\text{smiling} \doteq \text{no}\right] \\ & \vee \\ & X_2 : \text{robot} \left[\text{holding} \doteq \text{sword}\right] \end{aligned}$$

The other disjunct obtained by LINUS, [`smiling` = yes] $\wedge$ [`holding` = sword] is more specific than those obtained by INDIE. On the other hand, introducing the attribute `same-shape`, LINUS obtains that a robot belongs to *unfriendly* class if the attribute `same-shape` has as value *false*, i.e. body and head have not the same shape.

4.1.2. Drugs Dataset

The Drugs dataset contains the description of several drugs. This dataset has been used by the KLUSTER system (Kietz and Morik, 1994). From these descriptions KLUSTER can classify an instance in one of several classes, i.e. active substance, monodrug, sedative substance, etc. KLUSTER uses a representation language based on KL-ONE to represent both generalisations and domain objects. To use INDIE we have represented domain objects as feature terms. Descriptions obtained for these classes are similar to those obtained by KLUSTER. The main differences are due to the different representation formalism. A main semantic difference is that feature terms provide a *uniform* representation while description logics are *hybrid* since there are two different formalisms, one for describing concepts (T-box) and another one for describing instances (A-box). KLUSTER searches for a most specific generalisation (MSG) from positive examples. If MSG covers negative examples KLUSTER follows a particular algorithm to specialise MSG by means of introducing new `at-most` and `at-least` predicates in the feature descriptions. INDIE uses anti-unification to find a most specific description that subsumes positive examples (similarly to KLUSTER since both follow a bottom-up strategy). During specialisation INDIE introduces a disjunction of descriptions following the distance-based heuristic. INDIE obtains the following descriptions for the *monodrug* and *combidrug* classes:

$$\text{Monodrug} = X\text{: drug} \begin{bmatrix} \text{effects} \doteq Y\text{: drug - effect} \\ \text{contains} \doteq Z\text{: active - substance} \left[\text{affects} \doteq W\text{: symptom}\right] \end{bmatrix}$$

$$\text{Combidrug} = X\text{: drug} \begin{bmatrix} \text{contains} \doteq Y\text{: active - substance} \\ \qquad\qquad Z\text{ : active - substance} \end{bmatrix}$$

The description of the *monodrug* class means that a substance X is a monodrug when it contains a substance affecting some symptom and the substance X has some effect. A substance X is a *combidrug* when it has at least two active substances Y and Z. Notice that the main difference between both classes is using one active substance (*monodrug*) and more than one substance (*combidrug*). This fact derives from the definition of subsumption, namely that any example subsumed by the *combidrug* description above, needs to have (at least) two active substances for each feature `contains`. The *combidrug* examples would also be subsumed by *monodrug* description (since they also have one active substance) except that combidrugs do not satisfy the other two features (`effects` and `affects`).

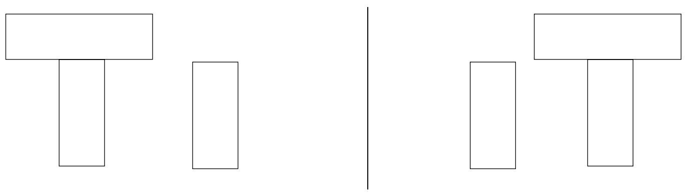


Figure 5.8. Two negative examples of *arch* covered by the description obtained by FOIL and LINUS.

4.1.3. Arch Dataset

This domain was introduced by Winston (1975). The Arch Dataset consists of two examples of the *arch* concept. Each arch is composed by three pieces (two vertical and one horizontal). As negative examples, there are two objects also composed of two vertical pieces and one horizontal piece but they do not form an arch (see appendix A). Authors that have used this domain (Quinlan, 1990; Lavrac and Dzeroski, 1994) represent background relations as Horn clauses. The arch description obtained by LINUS and FOIL (using both the closed world assumption) is the following:

arch(A,B,C) ← left-of(B,C), supports(B,A), not touches(B,C)

Notice that this description is consistent with the positive and negative examples provided to the system (see appendix A), but it also covers the unseen objects shown in figure 5.8 that are not arches. Thus, LINUS and FOIL need to include both objects as negative examples in order to obtain a correct description of *arch*. Using feature terms, INDIE obtains the following description:

$$\text{Arch} = X : \text{figure}\left[\text{left} \doteq Y : \text{brick}\left[\begin{array}{l} \text{left - to} \doteq T : \text{brick}\begin{bmatrix}\text{rigth - to} \doteq Y\\ \text{supports} \doteq Z\\ \text{touches} \doteq \text{no - one}\end{bmatrix}\\ \text{supports} \doteq Z : \text{block}\left[\text{over} \doteq \begin{array}{c}T\\Y\end{array}\right]\\ \text{touches} \doteq \text{no - one}\end{array}\right]\right]$$

i.e. an *arch* is an object X having a brick Y satisfying three conditions: 1) Y is left to a brick T, 2) Y supports a block Z, and 3) Y does not touch any brick (`touches` feature has value no-one). In turn, the block Z has to be supported by the blocks T and Y. Finally, brick T is to the right of Y, supports block Z and does not touch any brick. This description has been build using the four standard inputs. Notice that the description above obtained by INDIE is capable of obtaining a description that does not subsume the unseen negative examples in figure 5.8, since both vertical bricks in S have to support the central block Z. The graphical representation of the description obtained by INDIE (figure 5.9) makes explicit the relations among the pieces composing an arch.

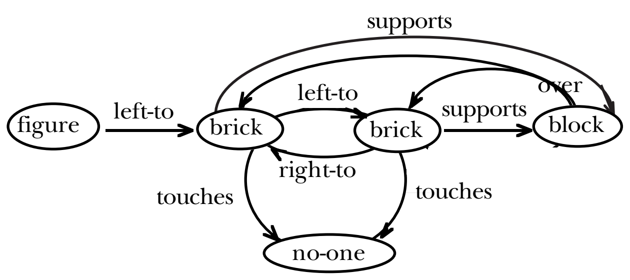


Figure 5.9. Graphical representation of the feature term obtained by INDIE to describe an arch.

4.1.4. Families Dataset

This domain, defined by Hinton (1989), consists of the definition of two families having twelve members each (see appendix A). Several ILP systems have been tested using this domain. In particular, LINUS obtains the following descriptions for the *mother* relation:

mother(A,B) ← daughter(B,A), not father(A,B) (1)
mother(A,B) ← son(B,A), not father(A,B) (2)

Rules obtained by FOIL to describe the *mother* relation are the following:

$$\text{mother(A,B)} \leftarrow \text{daughter(B,A), not father(A,C)} \quad (3)$$
$$\text{mother(A,B)} \leftarrow \text{son(B,A), not father(A,C)} \quad (4)$$

Notice that FOIL obtains more specific descriptions than LINUS since (3) and (4) use a new variable C. This new variable means that "A is not the father of anybody", whereas in descriptions (1) and (2) "A is not the father of B" (i.e. A should be the father of a person different than B). INDIE obtains the following description:

$$\text{mother} = X : \text{female}\left[\text{son} \doteq Y : \text{male}\right]$$

This description is equivalent to descriptions (2) and (4) above since the relation *not father* (used by LINUS and FOIL) is equivalent to define a person of sort *female* as INDIE does. On the other hand, INDIE obtains only one disjunct because the description obtained by anti-unification already subsumes all positive examples and does not subsume any negative example. After applying the simplification only the son feature remains. During the post-process, two features, son and daughter, have the same López de Mántaras distance; because of this any of them could have been eliminated, and INDIE has randomly chosen the elimination of the daughter feature.
We have also used INDIE to obtain a description for the *uncle* relation. The result is the following:

$$\text{uncle} = X : \text{male}\left[\text{niece} \doteq Y : \text{female}\right]$$

This description can be interpreted as "An uncle is a male that has at least a niece". As in obtaining the *mother* relation, during the post-process, two features, niece and nephew, have the same López de Mántaras distance, therefore INDIE has randomly chosen the elimination of the nephew feature.

4.1.5. Trains Dataset

This domain was introduced by Michalski (1980) to test the INDUCE system. Domain objects are 10 trains having different numbers of cars with various shapes carrying loads of different forms (see appendix A). The task is to distinguish between *eastbound* and *westbound* trains. Learners that use attribute-value representation have a great difficulty to solve this task due the variability in the number of structures and substructures present in the domain objects (for example, trains have a variable number of wagons). LINUS has to introduce an artificial variable to obtain the *eastbound* description (see details in Lavrac et al., 1991). Using ASSISTANT, LINUS obtains a description consisting of 19 Prolog clauses that after post-processing is reduced to one clause that is the same obtained by FOIL and INDUCE:

$$\text{eastbound}(A) \leftarrow \text{has-car}(A,B),\ \neg\ \text{long}(B),\ \neg\ \text{open-top}(B)$$

FOIL system obtains the following description for the *westbound* trains:

$$\text{westbound}(A) \leftarrow \text{has-car}(A,B),\ \text{long}(B),\ \text{2-wheels}(B),\ \neg\,\text{open-top}(B)$$

This description covers three of the five westbound trains. FOIL is not capable to obtain more descriptions due the encoding length heuristics. INDUCE can obtain a description covering all the westbound trains because uses constructive induction that introduces a new predicate: the number of wagons (car-count) of a train. Thus, *west* class are described by INDUCE as follows:

$$\begin{array}{l} \text{westbound}(A) \leftarrow \text{car-count}(A) = 3 \\ \text{westbound}(A) \leftarrow \text{has-car}(A,B),\ \text{jagged-top}(B) \end{array}$$

We have not data about the LINUS result in obtaining the *westbound* description. Using feature terms INDIE avoids the problem of the variability and is capable to obtain a description for both *eastbound* and *westbound* trains without introducing new predicates. The *eastbound* description obtained by INDIE is the disjunction of the following four feature terms:

$$\begin{array}{ll} \text{eastbound} = & X_1 : \text{train}\left[\text{wagon3} \doteq Y_1 : \text{closed-car}\left[\text{load-set} \doteq \text{circlelod}\right]\right] \\ & \vee \\ & X_2 : \text{train}\left[\text{wagon3} \doteq Y_2 : \text{closed-car}\left[\text{load-set} \doteq \text{rectanglod}\right]\right] \\ & \vee \\ & X_3 : \text{train}\left[\text{wagon3} \doteq Y_3 : \text{closed-car}\left[\text{load-set} \doteq \text{trianglod}\right]\right] \\ & \vee \\ & X_4 : \text{train}\left[\text{wagon3} \doteq Y_3 : \text{open-car}\left[\text{load-set} \doteq \text{hexagonlod}\right]\right] \end{array}$$

i.e. the *eastbound* trains are characterised by the load set of its third wagon, which can have `circlelod`, `rectanglod`, `trianglod` or `hexagonlod` form.

Notice that implicitly this description assumes that an eastbound train has at least three wagons. The *westbound* description obtained by INDIE is also a disjunction of three feature terms:

$$\begin{aligned}\text{westbound} = \; & X_1 : \text{train}\left[\text{wagon2} \doteq Y_1 : \text{open-car}\left[\text{form-car} \doteq \text{openrect}\right]\right] \\ \vee \; & \\ & X_2 : \text{train}\left[\text{wagon2} \doteq Y_2 : \text{open-car}\left[\text{form-car} \doteq \text{ushaped}\right]\right] \\ \vee \; & \\ & X_3 : \text{train}\left[\text{wagon3} \doteq Y_3 : \text{open-car}\left[\text{load-set} \doteq \text{rectanglod}\right]\right]\end{aligned}$$

Similarly to *eastbound* description, the `form-car` feature and the `load-set` feature are the more relevant to describe a west train.

4.1.6. Traffic Law Dataset

The Traffic Law dataset is concerned with some basic knowledge about traffic regulations in Germany. This dataset, composed of a set of twelve cases (see appendix A), has been used in the MOBAL system (Morik et al., 1993). Thus, from a set of predicates representing basic information about a traffic violation case, the involved vehicles and persons, and traffic regulations, the problem solving goal of MOBAL classifies of the case along several dimensions, i.e. determining who will be held responsible for violation, whether the responsible person will have to go to the court and how high the fine will be.

MOBAL represents each case by means of facts that provide information about objects and relations among objects (see appendix A). In addition to cases and topological background knowledge, MOBAL also has background knowledge of inferential relations in the domain. The background knowledge is usually represented in ILP as rules representing relations among concepts. As we have seen in section 1.2 of chapter 4, background knowledge using feature terms can be represented by means of sorts, feature path references and methods. In the Traffic Law dataset the background knowledge represented using feature terms is the following:

```
(define place)
(define (place no-parking-place)
        (no-parking true))
(define (no-parking-place Sidewalk))
(define (no-parking-place Bus-lane))
(define (place parking-place)
        (no-parking false))
(define (parking-place road-edge))

(define event
   (involved-vehicle )
   (parking-violation (>> no-parking car-parked involved-vehicle))
   ((unsafe-vehicle-violation (reify (>> major-corrosion involved-vehicle))
                              (reify (>> faulty-brakes involved-vehicle))
                              (reify (>> worn-tires involved-vehicle)))))
```

Since INDIE handles feature terms in normal form (see section 3.2.1 of chapter 3), path references and methods are reduced on demand by INDIE, and are converted to normal form.

We have used INDIE to find the descriptions of two traffic violations: *parking-violation* and *lights violation.* Nevertheless, we have not used the INDIE simplification post-process since we search for regularities among the positive examples (in a way similar to the MOBAL's module called RDT). In other words, we are interested in descriptions that do not subsume negative examples but having as much the features in common with the positive examples as possible.

Positive examples of *parking-violation* are those objects belonging to the `event` concept having the `parking-violation` feature with value *true.* Nevertheless, some traffic violation cases provided as input do not have this feature (see appendix A). The description for the *parking-violation* concept induced by INDIE is the following:

$$\text{X:traffic-case}\left[\text{event} \doteq \text{Y:event}\left[\begin{array}{l}\text{involved-vehicle} \doteq \text{Z: vehicle}\left[\begin{array}{l}\text{owner} \doteq \text{W}\\ \text{sedan} \doteq \text{true}\end{array}\right]\\ \\ \text{responsible} \doteq \text{W : person}\\ \text{car-parked} \doteq \text{P: no-parking-place}\end{array}\right]\right]$$

i.e. a parking violation occurs when the involved vehicle is parked in a place P that is of sort *no-parking-place.* The responsible is a person W who is also the owner of the vehicle. The description obtained by INDIE is equivalent to the following two rules:

$$\text{car-parked (X,Y)} \wedge \text{no-parking (Y)} \rightarrow \text{parking-violation(X)} \qquad (1)$$
$$\text{involved-vehicle (X,Y)} \wedge \text{owner (Z,Y)} \rightarrow \text{responsible (Z,X)} \qquad (2)$$

The rule (1) is used as background in MOBAL, and the rule (2) is discovered by this system using the cases of traffic violations and the background knowledge.

The description for the *lights-violation* concept obtained by INDIE is the following:

$$\text{X :traffic-case}\left[\begin{array}{l}\text{event} \doteq \text{Y :event}\left[\begin{array}{l}\text{involved-vehicle} \doteq \text{Z :vehicle}\left[\begin{array}{l}\text{owner} \doteq \text{W : person}\\ \text{sedan} \doteq \text{true}\\ \text{headlights-on} \doteq \text{false}\end{array}\right]\\ \text{time} \doteq \text{L :unlit-time}\end{array}\right]\\ \text{tvr}-\text{point s}-\text{P} \doteq \text{false}\end{array}\right]$$

i.e. a lights violation happens when the involved vehicle has the headlights off and the time is unlit. In this domain, `unlit-time` is a sort that has `dark-time` and `foggy-time` as subsorts. Thus, the description obtained by INDIE is equivalent to the following two rules used as background knowledge in MOBAL:

$$\text{time (X,Y)} \wedge \text{dark (Y)} \rightarrow \text{lights-necessary(X)}$$
$$\text{time (V,Y)} \wedge \text{place (V,Y)} \wedge \text{fog (X,Y)} \rightarrow \text{lights-necessary(X)}$$

Moreover, we have also used INDIE to learn the concepts of *court-citation* and *appeals*. INDIE obtains the same description for both concepts:

$$\text{X : traffic-case}\begin{bmatrix}\text{event} \doteq \text{Y : event}\begin{bmatrix}\text{involved-vehicle} \doteq \text{Z : vehicle}\begin{bmatrix}\text{owner} \doteq \text{W}\\ \text{sedan} \doteq \text{true}\end{bmatrix}\\ \text{responsible} \doteq \text{W : person}\end{bmatrix}\end{bmatrix}$$

i.e. *court-citation* and *appeals* are described by an `event` such that the person responsible of this event and the owner of the involved vehicle are the same person. MOBAL also obtains a strong relation between *appeals* and *court-citation* since both concepts occur in the same situations (see rules learned by MOBAL in appendix A). To confirm this strong relationship between both concepts, we have used INDIE with the simplification post-process obtaining the following descriptions:

$$\text{appeals} = \text{X : traffic-law}\left[\text{event} \doteq \text{Y : event}\left[\text{court-citation} \doteq \text{true}\right]\right]$$
$$\text{court-citation} = \text{X : traffic-law}\left[\text{event} \doteq \text{Y : event}\left[\text{appeals} \doteq \text{true}\right]\right]$$

i.e. the *appeals* concept is characterised by an `event` Y having the `court-citation` feature with value *true*. Conversely, the *court-citation* concept is characterised by an `event` Y having the `appeals` feature with value *true*.

4.1.7. The MESH dataset

The Mesh dataset is used to analyse stresses in physical structures, which are represented quantitatively as finite collections (meshes) of elements. Meshes can be classified as belonging to 17 solution classes. The GOLEM system has been applied to this domain to construct rules deciding on appropriate mesh resolution (Dolsak and Muggleton, 1992). An appropriate representation of the geometry of a structure must include the relations between its primitive components, which cannot be represented naturally in an attribute-value language. There is background knowledge describing some of the factors influencing the resolution of a mesh, such as types of edges, boundary conditions, and loadings. The predicates used by the GOLEM system and also the representation of these predicates using feature terms are shown in appendix A.

From three structures (a hydraulic press cylinder, a hook, and a paper mill) GOLEM obtains 56 different rules, some cover few examples and others are not useful in practice. For this reason, GOLEM authors have eliminated some non useful rules, selecting only 26 rules as useful. Nevertheless, they have also observed that if they allow the coverage of some negative examples, some of the remaining rules can be generalised. For instance (Dolsak and Muggleton, 1992), the following rule for the class *one* :

mesh (A,1) :- not-important (A), not loaded (A).

covers 18 positive examples and no negative examples. Instead, allowing cover a few negative examples, the following rule is induced:

mesh (A,1) :- not-important (A).

Since, this rule covers 27 positive examples and only one negative example, this second rule is preferred by the authors.

FOIL and LINUS systems, in addition to the three structures used by GOLEM, have also used two additional structures: a roller and a bearing box. Results obtained by both systems are similar to those of GOLEM. We have used the same three structures of GOLEM (hydraulic press cylinder, hook, and paper mill) to obtain the description for class *one*.

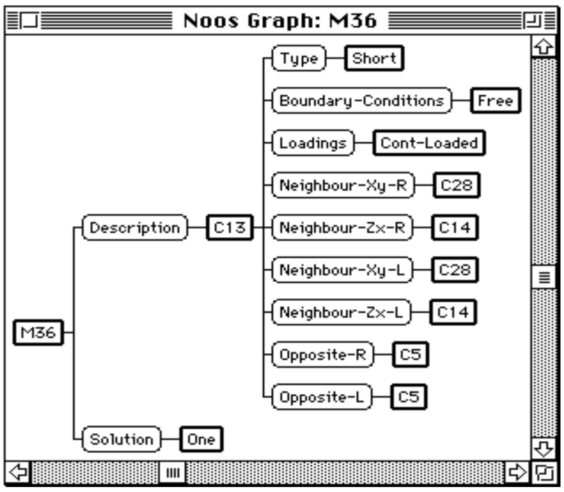


Figure 5.10. Representation of a Mesh domain object using feature terms.

Using feature terms, the objects of the Mesh dataset are represented as feature terms belonging to the sort *mesh-problem* (see figure 5.10). Objects in this sort have two features: `description` and `solution`. The `description` feature has as values objects belonging to the sort *edge*. An object of the *edge* sort can have a variable number of features (from 5 to 11 features). In Appendix A a complete description of the objects can be found. The values of the features (except `type`, `boundary-conditions`, and `loadings`) are *edges* that have, in turn, some of the mentioned features. The `solution` feature of *mesh-problem* objects contains the class to which the described *edge* can be classified. Meshes can be classified as belonging to 17 solution classes (from *one* to *seventeen*).

Each edge of a physical structure is related with many other edges that, in turn, are successively related with other edges. In this way, a very long chain of edges could be formed from each edge. Consequently, a description for a class may contain all the edges of a physical structure. For this reason, INDIE needs to control the depth of the descriptions in order to reduce the computation time and find simpler descriptions (i.e. descriptions relating an edge with its immediate neighbours). Depth control (see section 3.2) is a bias similar to those biases used in most of the ILP systems according to which rules using a smaller number of predicates and variables are preferred.

Using feature terms with maximum depth 1, INDIE obtains 36 descriptions for class *one*. These descriptions follow three kinds of patterns:

$$\begin{aligned}&X_1\text{: edge}\left[\text{neighbour-ZX-R} = \text{W : edge}\right]\\&\vee\\&X_2\text{: edge}\left[\text{neighbour-XY-R} = \text{V : edge}\right]\\&\vee\\&X_3\text{: edge}\left[\text{neighbour-YZ-R} = \text{S : edge}\right]\end{aligned}$$

where edges W, V, S have a concrete value in each description. Taking depth 2, the same kind of descriptions are obtained. Probably, to obtain more general descriptions we would need to allow higher levels of depth. We want to remark that FOIL, GOLEM and LINUS have some difficulties in dealing with the *neighbour* relations since they are not deemed as discriminant among positive and negative examples. This means that the *neighbour* relations do not appear in the obtained descriptions since authors report that their heuristics estimate a zero gain of information (Lavrac and Dzeroski, 1994). Nevertheless, this relation is proved important to take into account geometrical aspects of structures. In fact, Lavrac and Dzeroski (1994) encourage the use of this relation to improve the obtained results. This relation naturally appears in the descriptions obtained by INDIE since they are selected as relevant by heuristics from the set of features.

For class *two*, INDIE obtains 23 descriptions that follow the following patterns:

X_1: edge $\left[\text{type = long-for-hole}\right]$
$\vee$
X_2: edge $\left[\text{neighbour-ZX-R = V : edge}\right]$
$\vee$
X_3: edge $\left[\text{neighbour-YZ-R = W : edge}\right]$

where edges V and W are concrete edges such as, B28, A24, C8, etc. These descriptions are very different to those obtained by GOLEM, LINUS and FOIL (see appendix A).

4.1.8. Discussion

INDIE is capable to deal with relational domains in the formalism of feature terms. Our experiments show that INDIE gives results comparable to those obtained by FOIL, KLUSTER and LINUS in robots, drugs, families and arch domains. Notice that in the Robots domain INDIE obtains a relational description for *friendly* class (i.e. a robot belongs to the *friendly* class if it has the same shape of head and body). This same description is obtained by LINUS only after introducing a new feature representing the `same-shape` relation. INDIE also obtains best results than FOIL, INDUCE and LINUS when is applied to the arch domain, since INDIE does not need additional negative examples to find a correct description of arch.

Results obtained by INDIE over the trains domain are different from those obtained by FOIL, INDUCE and LINUS. One reason for the different descriptions for *eastbound* class is that INDIE does not use negation. On the other hand both systems, FOIL and INDUCE, have some difficulties in obtaining a description for the *westbound* class: FOIL cannot obtain a description covering the five westbound trains and INDUCE has to introduce a new predicate in order to achieve it. Instead, INDIE obtains three descriptions for *westbound* class without changing the initial representation of trains.

INDIE has also been applied to datasets as the Traffic Law dataset that uses background knowledge, and to the Mesh domain. Results of INDIE in both domains are comparable to those obtained by ILP systems as MOBAL, LINUS and mFOIL respectively.

4.2. Evaluation of the Accuracy

We have evaluated the accuracy of INDIE (using López de Mántaras heuristic) in order to assess its utility with respect to attribute-value learners of the decision tree family (based on similar heuristics). The INDIE method has been evaluated over Soybean (small and large) and Lymphography datasets from the Irvine ML Repository. Conditions used to evaluate them have been the same as those used by (Zhou and Dillon, 1995) in order to compare results. These conditions are the following:

- Attribute-value representation
- Training set containing the 67% of cases
- Test set containing the remaining 33% of cases

From this conditions we have randomly constructed 20 training sets and obtained the average of the results. During the evaluation we have observed that INDIE can provide two kinds of answers different to those provided by C4.5 (Quinlan, 1993) and CN2 (Clark and Niblett, 1989). On the one hand, INDIE can provide as answer "the example has not been classified", we call this situation a *no-solution* answer. On the other hand, INDIE can answer that there are several classes in which the current example can be classified, we call this situation *multiple solutions* answer. From now on, we speak of *no-solutions* and *multiple solutions* answer.

Depending on the application domain, answering multiple solutions may be not acceptable whereas in other domains multiple solutions may be a valuable information. For this reason, we will propose a new assessment of multiple solutions answer called *correctness*. In the next section we explain how the correctness of multiple solutions can be evaluated. We will also show that the accuracy definition is a particular case of correctness definition. Later the results obtained by INDIE for each dataset are explained in detail.

4.2.1. Analysis of Multiple Solution Answers

When a method answers only one solution, then this is either the correct solution or not. The accuracy is measured by the number of examples for which the method has produced the correct answer. Let A be the set of examples correctly solved. We can consider that A is a crisp set defined by a membership function μ_a from the set of examples E to the interval [0,1]:

$$\begin{cases} \mu_A(e) = 1 & \text{if } e \in \text{E has been correctly solved} \\ \mu_A(e) = 0 & \text{if } e \in \text{E has been incorrectly solved} \end{cases}$$

The measure of accuracy is the relative cardinality of A, i.e. the ratio of correctly solved examples with respect to the total number of examples. Therefore, if E is the set of training examples and E_C is the set of correctly solved examples (i.e. $\forall e_i \in E_C$ then $\mu_a(e_i) = 1$), then the accuracy can be computed using the following expression

$$\text{Accuracy} = \frac{\text{Card}(E_C)}{\text{Card}(E)}$$

We will now introduce the *correctness of a learning method*, an estimation that considers the set of examples correctly solved by a learning method as a fuzzy set. A fuzzy set C is defined by a membership function $\mu_C : X \rightarrow [0,1]$, where X is a universal set. The membership function μ_C indicates the membership degree to C of an element of X.

In our case, if E is the set of problems to be solved and sol(e_i) is the cardinality of the answer for a problem $e_i \in$ E, then we define X = {i | i = sol(e_i) $\forall e_i \in$ E}. Now, we define a membership function μ_C(i) that measures the degree to which an answer with i = sol(e_i) solutions is "correct". We want to assume that the more solutions the less informative is the answer. Thus, a correctness function μ_C(i) has to be a monotonically decreasing function such that:

- when the solution has only one correct answer (i=1) then $\mu_C(1)=1$
- when the solution has several answers (i > 1), all of them incorrect, then $\mu_C(i)=0$
- when the solution has as answers all the possible solution classes (i=N), then $\mu_C(N)=0$
- when the solution has several answers (1 < i < N) and one of them is the correct one, then $0 < \mu_C(i) < 1$

Obviously, when the method does not produce any solution for an example $\mu_C(0) = 0$. Nevertheless, this is a particular case and we are not interested in it. We can estimate the correctness of multiple answers taking the fuzzy membership function μ_C as a linear function (μ_1) or as a logarithmic function (μ_2) defined in the following way:

$$\mu_1(i) = \begin{cases} \frac{1}{1-N} i - \frac{N}{1-N} & \text{if } 0 < i \leq N \\ 0 & \text{otherwise} \end{cases}$$

$$\mu_2(i) = \begin{cases} \dfrac{\log_2 \frac{i}{N}}{\log_2 \frac{1}{N}} & \text{if } 0 < i \leq N \\ 0 & \text{otherwise} \end{cases}$$

The figure 5.11 shows the graphical representation of functions μ_1 and μ_2 for large Soybean (where N=19) and for the Lymphography (where N=4) domain. The graphic emphasises that logarithmic function μ_2 has a higher punishment form multiple solutions than the lineal function μ_1.

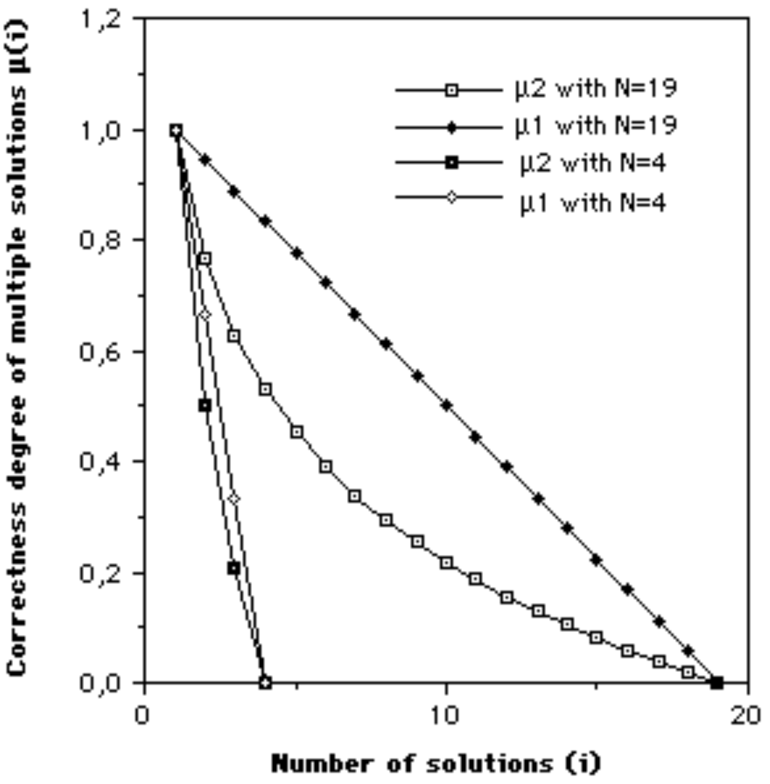


Figure 5.11. Representation of the correctness membership functions μ_1 and μ_2 for large Soybean and Lymphography domains, where N=19 and N=4 respectively.

We use the notion of *Σcount* to evaluate the correctness degree of INDIE. The *Σcount* is defined as follows:

- The *Σcount* of a fuzzy set C (Luca and Termini, 1972) is the summation of the membership degrees of all elements in C.

Applied over the set E of problems solved by a learning method, the *Σcount* gives us the summation of the correctness degree in which each problem in E has been solved. In other words, the *Σcount* of the set E is computed as follows

$$\sum_{e_i \in E} \mu_C(\text{sol}(e_i))$$

where $\text{sol}(e_i)$ is the number of solutions in the answer of the example e_i.

The *Σcount* can be normalised in the interval [0,1] by defining the *relative count* (or membership to the *relative cardinality*) of a set E as follows:

$$G(E) = \frac{\sum_{e_i \in E} \mu_C(\mathrm{sol}(e_i))}{\mathrm{Card}(E)}$$

We call G the *correctness of a learning method.*

Notice that G(E) is the accuracy when the set E is crisp (since $\mu_C(1) = 1$ and 0 otherwise). In other words, accuracy is the special case of correctness when the "correct" set is crisp instead of fuzzy. G produces as result a number in the interval [0,1] (where better methods have G close to 1).

Thus the performance of a learning method can be assessed using the relative count function (using either of the correctness membership functions μ_1 or μ_2 described above). Summarising, in the next sections we use the following correctness functions

$$G_1(E) = \frac{\sum_{e_i \in E} \mu_1(\mathrm{sol}(e_i))}{\mathrm{Card}(E)} \quad \text{and} \quad G_2(E) = \frac{\sum_{e_i \in E} \mu_2(\mathrm{sol}(e_i))}{\mathrm{Card}(E)}$$

to evaluate INDIE's result on several standard datasets.

DOMAIN	Method	Contains Correct answer	No solution	Multiple solutions	$G_1(E)$	$G_2(E)$
Soybean (small)	INDIE	99%	1%	4%	0,977	0,970
Soybean (large)	INDIE	90,5%	4,1%	11,3%	0,899	0,879
	CN2	81,6%	—	—	0,816	0,816
	C4.5	80%	—	—	0,800	0,800
Lymphography	INDIE	79,9%	7,9%	11,6%	0,765	0,742
	CN2	81,7%	—	—	0,817	0,817
	C4.5	76,4%	—	—	0,764	0,764

Table 5.1. Results of the INDIE method compared to CN2, and C4.5 using both (small and large) Soybean and the Lymphography datasets. Results of the propositional learners have been obtained from (Zhou and Dillon, 1995).

4.2.2. Correctness Results in the Discrimination Task

Table 5.1 shows results of INDIE compared to those of the propositional learners CN2 and C4.5. Used over the large Soybean dataset, INDIE has $G_1(E) = 0{,}899$ and $G_2(E) = 0{,}879$ of correctness. INDIE produces a 11,3% of multiple solutions answers and a 4,1% of no solution answers. Column labelled as "contains correct answer" in table 5.1 is the percentage of answers containing the correct solutions. Thus, if we consider that any answer containing the correct solution is a correct answer, then INDIE provides an accuracy of the 90% over large Soybean.

Using INDIE over the Lymphography dataset produces a $G_1(E)$ = 0,768 and $G_2(E)$ = 0,742 of correctness, with 7,9% of no solution answers and 11,6% of multiple solution answers. In column "contains correct answer" there is the percentage of accuracy considering that the multiple solution answers that contain the correct solution are correct answers.

Notice that $G_1(E)$ and $G_2(E)$ provide a measure near to C4.5 but worse than CN2 for the Lymphography dataset. A detailed analysis of the multiplicity of answers (see figure 5.12) reveals that most of the times (a 95,6%) INDIE provides two solutions. Thus, a first conclusion from these results over large Soybean and Lymphography datasets, is that the INDIE behaviour is comparable to those of the propositional learners.

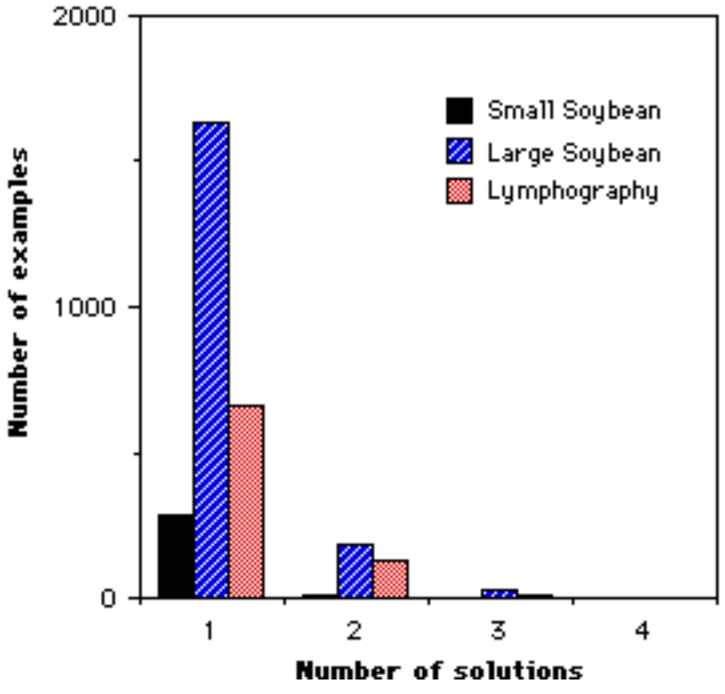


Figure 5.12. Analysis of multiple solutions obtained by INDIE.

In order to analyse the difference in the behaviour of INDIE whether or not the domain objects have features with unknown values, we have evaluated INDIE over the small Soybean dataset. The small Soybean dataset shares a set of 49 examples with large Soybean dataset, but they have no feature with an unknown value. This means that all the examples in small Soybean dataset are described by the same set of features. Table 5.1 also shows that in small Soybean dataset, INDIE has $G_1(E) = 0{,}977$ and $G_2(E) = 0{,}970$ of correctness, with low percentages of no solution and multiple solution answers.

4.2.3. Discussion

The behaviour of INDIE over propositional domains is comparable to the behaviour exhibited by propositional learners such as C4.5 and CN2. Nevertheless there are some differences between the results of the propositional learners and those of INDIE. A first difference is that propositional learners have a special mechanism to deal with imperfect data (unknown and noisy values). INDIE can deal with unknown values as a consequence of the feature term representation. On the other hand, INDIE has not special mechanism handling noisy values. Probably a treatment for imperfect data would improve the INDIE results.

A second difference is that propositional learners provides only one (correct or incorrect) solution, whereas INDIE can produce either multiple solution answers or no solution answer. In order to compare the INDIE results with those of the propositional learners, we have introduced a correctness function. The accuracy measure currently used to evaluate propositional learners is a particular case of the correctness function.

Commonly, propositional learners use examples represented as vectors of attribute-value pairs. In our experiments with INDIE, examples have been represented using feature terms. Nevertheless, we have not used all the power of this formalism since examples of standard datasets have not been represented in a structured way. The reason is that we do not have the domain knowledge required to translate them, in a meaningful way, to a structured representation. So we represent vectors by means of feature terms.

5. Conclusions

INDIE is a heuristic bottom-up inductive learning method that uses feature terms to represent domain objects and background knowledge. Typically, inductive learning methods are evaluated on different datasets according to whether they are propositional or relational. INDIE can be applied to both kinds of datasets although our main concern has been on relational problems, where feature terms may be more useful.

The INDIE performance over relational datasets is similar to the performance of relational learners such as FOIL, LINUS and GOLEM. Moreover, INDIE performance over propositional datasets is also similar to propositional learners such as C4.5 and CN2.

Feature terms allow a natural representation of partially described domain objects, i.e. having features with unknown values. Typically, when an example has an attribute with unknown value, some value has to be assigned to this attribute. INDIE has another, more simple mechanism to deal with imperfect data, thanks to the feature term representation. Feature terms support descriptions of objects with partial (incomplete) information. A feature with an unknown value is formally modelled simply as having as value *any*, the zero-information sort. In practice, this feature is simply not considered in the description of that example. Therefore, using feature terms is not necessary to assign an artifactual value to an unknown feature.

However, some aspects related to unknown values can be improved in INDIE. The description D obtained from the anti-unification of the positive examples E^+ of a class C contains all the features common to all the examples in E^+. If a feature A has an unknown value in some examples in E^+, then A will not appear in D. As a consequence, D is more general than the description obtained without having unknown values. Therefore, unknown values tends to produce an increment of the number of multiple solution answers produced by INDIE (since the description of an unseen example may be subsumed by the description of several classes).

Concerning noisy values, they only affect the INDIE results when they are common to all the positive examples (E^+) of a class, since a non-common feature does not appear due to the anti-unification operation. A noisy value v of a feature A only influences the result if it belongs to a sort S different to the sorts to which the remaining values of A belong. As a consequence, INDIE makes a generalisation for A that is unnecessary.

Thus, the main consequence of noisy data is that unnecessary generalisations are produced. These unnecessary generalisations can produce, in turn, an overgeneralisation of the obtained descriptions. The overgeneralisation is solved in INDIE by specialisation, that in turn introduces more disjunctions of descriptions. The INDIE post-process eliminates those features that are not relevant for describing a class. Thus, the INDIE post-process can contribute to the elimination of the noisy effects when a feature whose value has been over-generalised due to a noisy value is considered as irrelevant.

As future work we propose to design a mechanism to deal with imperfect data. In particular, mechanisms similar to those used in ILP learners, such as FOIL (Quinlan, 1990) or GOLEM (Dolsak and Muggleton, 1990), or some propositional learners, such as CN2 (Clark and Niblett, 1989) could be used. Those mechanisms search for descriptions that are not completely discriminant. FOIL can obtain clauses that do not cover all the positive examples. Instead, GOLEM builds clauses that can cover a pre-determined number of negative examples. Clauses built by CN2 can cover some negative examples. Any of these threshold mechanisms could be incorporated to INDIE.

Chapter 6

The DISC method

1. Introduction

In this chapter we describe DISC, a heuristic top-down method for feature terms induction. DISC starts with the top feature term *any* (that is the most general term) and specialises it whenever negative examples are subsumed. The contributions of DISC are the representation of examples and hypothesis as feature terms, and the use of the anti-unification concept as bias to specialise descriptions.

In the next section a general view of the DISC method in the discrimination task is described. Then, in section 3 the DISC algorithm is described in detail. Finally, in section 4, we provide results of the DISC evaluation over the same domains as INDIE.

2. General View of DISC

The goal of DISC in the discrimination task is to obtain a discriminant description for a solution class C_k. The obtained description has to be a most general description that subsumes positive examples and does not subsume negative examples (see the description of induction in chapter 4). As all the top-down methods, this discrimination description is obtained by specialising a current description that covers both positive and negative examples until no negative examples are covered. As in INDIE, positive examples of the solution class C_k are those training examples classified as belonging to C_k. Negative examples of C_k are those belonging to solution classes different than C_k. We assume that the training examples are correctly classified.

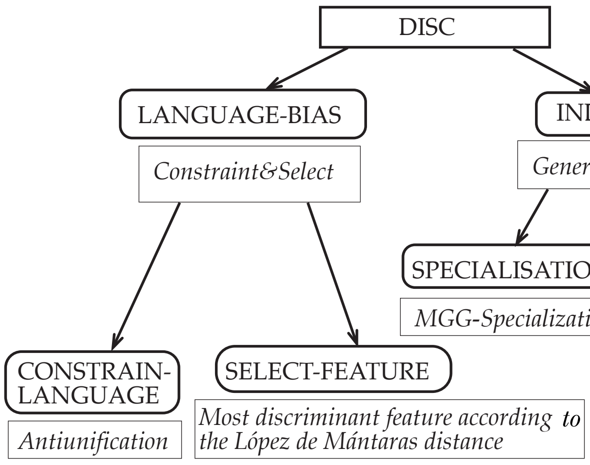


Figure 6.1. Decomposition of DISC method.

The DISC method used in the discrimination task is composed by two tasks (figure 6.1): `Language-bias` and `Induction`. The `Language-bias` task has as goal to select a feature to be used to specialise the current concept description. The method `Constraint&Select` in this task is decomposed in two subtasks: `Constraint-language` and `Select-feature`. The `Constraint-language` task determines a subset of candidate features to specialise the current description. Let D be the description obtained from the anti-unification of the positive examples of the class C_k. DISC considers as candidates to specialise the current description those features that are leaves of D. From this set, the `Select-feature` task chooses the most relevant feature using the López de Mántaras distance.

Once the `Language-bias` task has selected an appropriate feature to specialise the current concept description D, the `Induction` task has to add this feature to D. The method used by `Induction` task, called `General-to-Specific`, decomposes in two subtasks: `specialisation` and `Discriminant-description`. The `specialisation` task adds the selected feature to the current concept description. The `discriminant-description` task checks if the resulting description subsumes some negative examples. If the specialised description D still subsumes some negative example, the `discriminant-description` task uses the DISC method to newly specialise the obtained description. Otherwise the description D is given as result. In the next section the DISC algorithm is explained in detail.

Examples and concept descriptions are represented as feature terms, therefore DISC handles unknown values as INDIE: features having unknown value do not appear in the description of examples since they have value *any* (see section 3.3 of chapter 4).

3. Description of the DISC Algorithm

Given a set of training examples E = {e_1, ..., e_m} and a set of solution classes C = {C_1, ..., C_n}, the goal of DISC is to obtain a discriminant description D_k for each solution class C_k. Each example e_i is a feature term having a subset of features A_i = {A_{i1}, ..., A_{in} | $A_{ij} \in F$}. Each training example can have a different subset of the legal feature set *F*. Each feature A_i has as value a set of objects O_i where each $o_{ij} \in O_i$ is a feature term that can have, in turn, a subset of features in F. Background knowledge can be used by the DISC method in the same way as in INDIE (see section 1.2 in chapter 4).

A disjunctive description D = {D_k} represents a disjunction of feature term descriptions for the current solution class C_k. Each D_k is a most general description subsuming a subset of positive examples of C_k and not subsuming negative examples. In the discriminant task, negative examples of a solution class C_k are all those training examples that do not belong to C_k.

```
Initialisation: D = D_j = any; S_j = E^+
Function DISC (S_j, E^-, D_j)
    if there is some e ∈ E^- such that D_k ⊑ e
        then   d_a = anti-unification (S_j)
               A_l =  {A_i | features of d_a chosen according to bias}
               A_d = Select-feature (A_l, S_j)
               P_dk =  partition induced by A_d having minimum LDM distance
               for each set S_i ∈ P_d do
                   A_i = Path containing the feature A_d in d_a
                   D_i = Specialise (D_j, A_i)
                   Add DISC(S_i, E^-, D_i) to D
               end-for
        else return D_k
    end-if
end-function
```

Figure 6.2 DISC function that obtains a disjunction of descriptions for the current class. Words in bold are processes explained in the sections 3.1, 3.2 and 3.3 respectively.

Given a set of positive examples E^+ for a solution class C_k, let E^- be the set of negative examples for C_k. The goal of DISC is to build a discriminant description D for the solution class C_k. The strategy of the DISC algorithm (figure 6.2) is to specialise D until no negative examples are subsumed. Initially D = {D_k} = *any*, i.e. D has only one disjunct, say D_k, that is the description D_k = *any* (where *any* is the top element of the sort lattice). In such situation D_k subsumes all the positive examples E^+, so it has to be specialised.

Let us suppose now that the current hypothesis is $D_j \in$ D subsuming some negative example. Let S_j be the subset of positive examples subsumed by D_j. In such situation, DISC specialises D_j using a top-down strategy. Features candidates to specialise D_j are those belonging to the leaves of the feature term d_a obtained from the anti-unification of the examples in S_j. The specialisation of a description D_j is made by selecting the most discriminant feature a_d of those contained in the leaves of d_a. Each feature induces (1) a partition according to the values that it takes in the examples of S_j and also (2) other partitions taking into account the sort hierarchies of those values (see section 3.3 in chapter 5). The heuristic takes as the most discriminant feature the one inducing a partition having the minimum López de Mántaras (LDM) distance to the correct partition.

Let a_d be the most discriminant feature and P_{dk} the partition having minimum distance to the correct partition. In this situation, for each set S_k of the partition P_{dk} a new description D_{jk} is build by specialising the current description D_j. The DISC algorithm has to be recursively applied taking as parameters the set S_k, the specialisation D_{ki} and the set of negative examples E^-.

In the next sections we describe the main steps of DISC, i.e. which is the bias used to specialise a description, how select the appropriate feature to make the specialisation and how the specialisation is performed.

3.1. Bias

Let us assume that a current concept description D_i subsumes a subset $S \subseteq E^+$ of positive examples and also subsumes some negative examples $N \subseteq E^-$. Any feature included in the description of some positive example in S would be a good candidate to specialise the description D_i. Nevertheless, using feature terms, examples can be described by features that are not present in all examples. For instance, let us consider the following feature terms E1 and E2 describing two marine sponges specimens:

$$E1 = X:\text{sponge}\left[\begin{array}{l}
\text{skel} \doteq Y:\text{spic - skel}\left[\begin{array}{l}
\text{size} \doteq \begin{array}{r} Z:\text{megas}\left[\begin{array}{l}\text{axis} \doteq \begin{array}{l}\text{four}\\ \text{one}\end{array}\\ \text{megas} \doteq \begin{array}{l}\text{triaena}\\ \text{oxea}\end{array}\end{array}\right] \\ P:\text{micros}\left[\text{micros} \doteq \begin{array}{l}\text{oxyaster}\\ \text{chiaster}\\ \text{sterraster}\end{array}\right]\end{array}\\
\text{spicarch} \doteq \text{radiate}
\end{array}\right]\\
\text{quim} \doteq \text{silica}\\
\text{grow} \doteq \text{massive}\\
\text{dot} \doteq \text{no}\\
\text{hollow} \doteq \text{no}\\
\text{macro} \doteq \text{none}
\end{array}\right]$$

$$E\,2 = X:\ \text{sponge}\begin{bmatrix} \text{skel} \doteq Y:\ \text{spic - skel}\begin{bmatrix} \text{size} \doteq Z:\ \text{megas}\begin{bmatrix} \text{axis} \doteq \begin{matrix}\text{four}\\ \text{one}\end{matrix} \\ \text{megas} \doteq \begin{matrix}\text{dichotriaena}\\ \text{oxea}\\ \text{orthotriaena}\end{matrix} \end{bmatrix} \\ \text{P: micros}\begin{bmatrix}\text{micros} \doteq \text{sterraster}\end{bmatrix} \end{bmatrix} \\ \text{quim} \doteq \text{silica} \\ \text{form} \doteq \text{irregularly - massive} \\ \text{grow} \doteq \text{massive} \\ \text{hollow} \doteq \text{yes} \\ \text{osc} \doteq \text{yes} \end{bmatrix}$$

Notice that E1 and E2 have some features in common (for instance `axis`, `megas`, `micros`, `size`) but there also are features that only appear in one of the descriptions. For instance, `spicarch` and `macro` appear in E1 but not in E2, and the feature `osc` is in E2 but not in E1. If DISC would select the feature `osc` (appearing only in the description E2) to specialise the current description D_i, the obtained specialisation of D_i will not subsume E1 since E1 does not contain the feature `osc`. In order to avoid this problem and reduce the set of possible candidate features, DISC applies a bias based on anti-unification. The bias consists of taking as candidates to specialise D_i those features contained in the feature term obtained from the anti-unification of the positive examples. This bias reduces considerably the set of candidate features. Moreover it is easily applicable since the anti-unification provides all that is common for a set of examples. In our example, without applying the bias, the set of feature candidates to specialise the current description is the set {`axis`, `megas`, `micros`, `size`, `spicarch`, `skel`, `quim`, `grow`, `dot`, `hollow`, `macro`, `form`, `osc`}. By applying the bias, the anti-unification of E1 and E2 is the following description:

$$X:\ \text{sponge}\begin{bmatrix} \text{skel} \doteq Y:\ \text{spic - skel}\begin{bmatrix} \text{size} \doteq Z:\ \text{megas}\begin{bmatrix} \text{axis} \doteq \begin{matrix}\text{four}\\ \text{one}\end{matrix} \\ \text{megas} \doteq \begin{matrix}\text{triaena}\\ \text{oxea}\end{matrix} \end{bmatrix} \\ \text{P: micros}\begin{bmatrix}\text{micros} \doteq \text{sterraster}\end{bmatrix} \end{bmatrix} \\ \text{quim} \doteq \text{silica} \\ \text{grow} \doteq \text{massive} \\ \text{hollow} \doteq B:\ \text{boolean} \end{bmatrix}$$

Therefore, the set C of feature candidates to specialise D_i is C = {`axis`, `megas`, `micros`, `size`, `skel`, `quim`, `grow`, `hollow`}.

In order to further reduce the set of candidate features to specialise a description, we apply a second bias. This second bias is the same one used in INDIE, i.e. taking only the features belonging to the leaves of the feature term obtained from the anti-unification of the positive examples in S. Applying this second bias to our example, DISC candidate features for specialisation are {`axis, megas, quim, grow, hollow`}. Any of the features contained in this set could be used by DISC to specialise the description D_i.

As in INDIE, there is a parameter (given by the user) called *maximum depth* that controls the depth of a feature term (see section 3.2 in chapter 5). Such parameter allows the determination of the leaves of a feature term *f* when *f* has features with circular references between their values.

3.2. Selection of the Most Discriminant Feature

Given a description D_i that subsumes some negative examples, the goal of the `select-feature` task is to obtain the most discriminant feature a_d. The main idea pursued in taking the most discriminant feature is to build descriptions with a small number of features. This is the same idea followed by most of learners that search for short descriptions of concepts. As will be seen in the evaluation of DISC, the obtained descriptions are similar to those obtained by the INDIE method.

Let S_i be the subset of examples subsumed by the current description D_i and A_l be the set of features candidate to specialise the current description. Each feature $a_i \in A_l$ induces a set of partitions $\{P_{i1},\dots P_{ik}\}$ on the training set. Each partition P_{ij} is induced according to different combinations of the sorts to which the possible values of a_i in S_i belong (see section 3.3 in chapter 5). We can measure the distance of each induced partition to a partition P_c that is a restriction of the correct partition[1]. Using López de Mántaras distance, DISC selects the attribute $a_{min} \in A_l$ inducing a partition $P_{min} \in \{P_{i1},\dots P_{ik}\}$ having the least distance to the correct partition. This heuristic chooses a_{min} as the most discriminant feature inside A_l.

```
Function Select-feature ($A_l$,$S_j$)
  for each $a_i \in A_l$ do
     Let {$P_{i1}$,… $P_{ik}$} be the set of partitions induced by $a_i$ in $S_i$
     for each $P_{ij} \in$ {$P_{i1}$,… $P_{ik}$} do
        d($P_{ij}$,$P_c$)  ;; López de Mántaras distance to the correct partition $P_c$
  end-for
  return $a_{min}$ := $a_i$ with $P_{ij}$ such that $d_{min}$ = min {d($P_{ij}$,$P_c$)}
end-function
```

Figure 6.3. The `select-feature` algorithm of the DISC method.

[1] Notice that now the set of positive examples is S_i.

3.3. Specialising a description

Let D_i be the description to specialise, S_i the subset of positive examples subsumed by D_i, A_l the set of features candidate to specialise D_i, $a_{min} \in A_l$ the most discriminant feature and $P_{min} = (S_{min,1} \dots S_{min,N})$ the partition induced by a_{min} having minimum López de Mántaras distance to the correct partition. That is to say, all examples contained in each $S_{min,j}$ have the same value or sort in the feature a_{min}. Thus, for each partition set $S_{min,j}$ a specialisation of D_i, that we call D_{ij}, is build. Each D_{ij} has the same features as D_i plus the path from the root of d_a to the feature a_{min} and where a_{min} has as value the one taken by a_{min} in $S_{min,i}$.

Notice that the selection of a leaf feature introduces in the description D_i all the features in the path from the root to the selected feature. For example, let us suppose that $D = \{D_i\} =$ *any* and that the feature `sterr` (see figure 5.3 in chapter 5) is selected as the most discriminant. The `sterr` feature takes two values: *flat* and *globular*. Therefore the description D_i will be replaced by the disjunction of two descriptions that are specialisations of D_i. Both specialisations contain the path formed by the features `skel`, `size`, `micros` and `sterr`. Figure 6.4 shows one of these specialisations. The other has *globular* as value of the `sterr` feature.

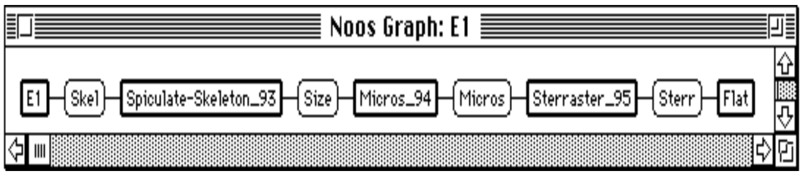


Figure 6.4. The path from the root to the *sterr* feature.

Therefore the overgeneral description D_i is specialised into $D'_i = \{D_{ik}\}$ ($k = 1 \dots N$) where D'_i is a disjunction of N descriptions, and N is the number of sets contained in the partition P_{min} associated to the discriminant feature a_{min}.

4. Evaluation of DISC

As in INDIE, two aspects of the DISC method have been evaluated: the suitability of inductively created descriptions and its correctness. Suitability of predictions have been evaluated over several relational domains. Predictivity has been evaluated over Lymphography and Soybean databases which can be used to evaluate propositional learners. In the next sections both aspects are explained (see appendix A for the description of domains). In chapter 9 we also analyse the DISC behaviour when applied to marine sponges classification.

4.1. Evaluation of the Descriptions Suitability

In this section we analyse DISC as a concept learner over several relational domains: robots, drugs, arch, trains and traffic law (see Appendix A and also INDIE in previous chapter for their definition). Results obtained by DISC are compared to results obtained by some ILP systems (mainly FOIL, LINUS and KLUSTER) in the same domains.

4.1.1. Robots Dataset

The domain of Robots (Lavrac and Dzeroski, 1994) contains a description of six robots. Each robot is described by five features: `smiling`, `holding`, `has-tie`, `body-shape` and `head-shape` and they can belong to two solution classes: *friendly* and *unfriendly* (see appendix A). LINUS describes robots using an attribute-value representation whereas in DISC robots are described using feature terms. The following descriptions for the *friendly* class are obtained using DISC:

$$\begin{aligned} \text{Friendly} &= X_1 : \text{robot}\left[\text{holding} \doteq \text{balloon}\right] \\ &\vee \\ &X_2 : \text{robot}\begin{bmatrix}\text{holding} \doteq \text{flag} \\ \text{has - tie} \doteq \text{yes}\end{bmatrix}\end{aligned}$$

i.e. a robot belongs to the *friendly* class when 1) it holds a balloon, or 2) it holds a flag and has tie. The obtained descriptions show that the most discriminant feature was `holding` since it appears in both disjuncts. `Holding` has two possible values in the positive examples *balloon* and *flag*. Thus two descriptions are created, X_1 and X'_2, both having only the feature `holding`. The value *balloon* is discriminant enough for X_1, since there is no negative example having this value, so the description X_1 is already correct. However, the description X'_2 having the feature `holding` with value *flag* subsumes one negative example (R6, see figure 5.7 in previous chapter), therefore DISC is recursively applied over the description X'_2. Following the same process, DISC finds that the most discriminant features covering the example R2 are `has-tie` and `smiling`. Both features can take two values: *yes* and *no* and have the same value of the López de Mántaras distance. In other words, both features are equally discriminant and DISC randomly chooses `has-tie` to specialise X'_2. This new specialisation would obtain two new descriptions (both specialisations of X'_2). Nevertheless, the feature `has-tie` only takes the value *yes* in the positive examples so only one description (the X_2 above) is generated.

As it has been explained in section 3.1.1. of chapter 5, LINUS obtains the following descriptions for the friendly class:

$$\text{Class} = \text{friendly } \textbf{if } [\text{smiling} = \text{yes}] \wedge [\text{holding} = \text{balloon}] \quad (1)$$

$$\text{Class} = \text{friendly } \textbf{if } [\text{smiling} = \text{yes}] \wedge [\text{holding} = \text{flag}] \qquad (2)$$

Notice that the description X_1 obtained by DISC is more general than the description (1) obtained by LINUS. On the other hand, X_2 and description (2) are equivalent since DISC considers equally discriminant the features `has-tie` and `smiling`.

The description obtained by DISC for the *unfriendly* class is the following:

$$\begin{aligned} \text{Unfriendly} = {} & X_1 : \text{robot}\left[\text{holding} \doteq \text{sword}\right] \\ & \vee \\ & X_2 : \text{robot}\begin{bmatrix}\text{holding} \doteq \text{flag} \\ \text{has-tie} \doteq \text{no}\end{bmatrix} \end{aligned}$$

As it has been shown in section 3.1.1. of previous chapter, LINUS obtains the following descriptions for the *unfriendly* class:

$$\text{Class} = \text{unfriendly } \textbf{if } [\text{smiling} = \text{no}]$$
$$\text{Class} = \text{unfriendly } \textbf{if } [\text{smiling} = \text{yes}] \wedge [\text{holding} = \text{sword}]$$

Comparing DISC and LINUS results, we see that they are equivalent. Notice that the descriptions of both classes, *friendly* and *unfriendly*, are more similar among them than those obtained by INDIE. This is due to the fact that INDIE focuses in the whole feature term structure and, consequently, it can detect path equality (as occurs in the *friendly* class description). Instead DISC searches each time only for one discriminant feature to specialise the current hypothesis.

4.1.2. Drugs Dataset

The Drugs domain used in KLUSTER (Kietz and Morik, 1994) consists of descriptions of some drugs that can be classified as: *monodrug, combidrug, placebo, active, additive, sedative,* etc. The DISC method, using feature terms, has produced the following description for the *monodrug* class:

$$\begin{aligned} \text{Monodrug} = {} & X_1 : \text{drug}\left[\text{contains} \doteq \text{Active-substance}\left[\text{affects} \doteq \text{headache}\right]\right] \\ & \vee \\ & X_2 : \text{drug}\begin{bmatrix}\text{contains} \doteq \text{Active-substance}\left[\text{affects} \doteq \text{stress}\right] \\ \text{effects} \doteq \text{sedative}\end{bmatrix} \end{aligned}$$

i.e. a *monodrug* can be defined in two ways: 1) as a active substance affecting to the headache, or 2) as a active substance with sedative effects affecting the stress. Thus, `affects` is the most discriminant feature but there is a combidrug that also affects the stress and, consequently, the obtained description has to be specialised by adding the `affects` feature.

DISC obtains the following descriptions for the *combidrug* class:

$$\begin{aligned}\text{Combidrug} &= X_1 : \text{drug}\left[\text{contains} \doteq \begin{matrix}\text{Phenazetin}\\ \text{Prophymazon}\\ \text{Nhc}\end{matrix}\right]\\ &\vee\\ &\quad X_{\,2} : \text{drug}\left[\text{contains} \doteq \begin{matrix}\text{Oxazepun}\\ \text{Finalin}\end{matrix}\right]\end{aligned}$$

There are two substances defined as combidrug. In searching for the description of the *monodrug* class, DISC takes the feature `contains` as the most discriminant. `Contains` has as values in the examples two sets, so two descriptions X_1 and X_2 have been obtained. None of both descriptions subsume negative examples.

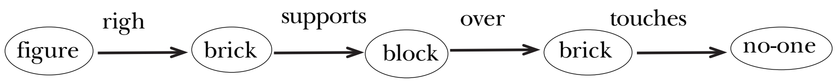


Figure 6.3. Path to be included in the description of *arch* due to the leaf bias. The heuristic selects the `touches` feature as the most discriminant.

4.1.3. Arch Dataset

This domain, introduced by Winston (1975) has as input the description of four objects (two arches and two non-arches). An arch is a *figure* formed by three pieces: two vertical and one horizontal. We have applied DISC to obtain the description of the *arch* concept. The heuristic has selected the feature `touches`. `Touches` is a leaf of the description obtained from the anti-unification of the examples of the *arch* concept. This implies the inclusion in the *arch* description of the path from the root of the anti-unification to the `touches` feature. In other words, the path composed by the features `right`, `supports` and `over` is added to the *arch* description concept (see figure 6.3). This instance shows the utility of the leaf bias. Therefore, DISC obtains the following description:

$$\text{Arch} = \text{X:figure}\left[\text{rigth} \doteq \text{Y:brick}\left[\text{supports} \doteq \text{Z:brick}\left[\text{over} \doteq \text{T:brick}\left[\text{touches} \doteq \text{no - one}\right]\right]\right]\right]$$

That is to say, an *arch* is a complex of objects (a *figure*) X having a brick Y at the right supporting a brick Z. This brick Z is over another brick T which does not touch any other brick. Notice that the description obtained by DISC does not cover the negative example of the figure 5.8 in chapter 5 (as is the case with the descriptions obtained by LINUS and FOIL).

4.1.4. Families Dataset

This domain of Family relations was defined by Hinton (1989). This dataset contains the description of two families with twelve members each (see appendix A). We have represented the examples in this dataset as feature terms belonging to the sort *person*. The sort *person* has two subsorts *male* and *female*. Thus, members of a family are defined as belonging to either *male* or *female* subsorts. Features of the sort person correspond to familiar relationships as `mother, father, brother, sister`, etc. The goal of DISC in using this domain (as LINUS) is to obtain a description for the concept *mother*. The obtained description has been the following:

mother = X: female [daughter = Y : female]

In searching for the most discriminant feature, DISC obtains two features equally discriminant: daughter and son. Now, the feature daughter has been randomly chosen, nevertheless DISC could also have obtained the following description:

mother = X: female [son = Z : male]

Both descriptions were also same obtained by INDIE, which are in turn equivalent to those obtained by FOIL and LINUS (see section 4.1.4 in previous chapter).

We have also used DISC to obtain the description for the *uncle* relation. As in INDIE we have obtained the following description:

uncle = X: male [nephew = Y : male]

that is equivalent to the description:

uncle = X: male [niece = Z : female]

since `nephew` and `niece` are equally discriminant.

4.1.5. Trains Dataset

This domain, introduced by Michalski (1980), has as training examples the description of ten trains. Five of these trains have eastward direction and the remaining five trains have westward direction. Thus, there are two solutions classes to characterise: *eastbound* and *westbound*. DISC obtains the following descriptions for the *eastbound* trains:

$$\begin{aligned}
\text{eastbound} = {} & X_1 : \text{train}\left[\text{wagon}3 \doteq Y_1 : \text{closed-car}\left[\text{load-set} \doteq \begin{matrix}\text{circlelod}\\ \text{circlelod}\end{matrix}\right]\right]\\
& \vee\\
& X_2 : \text{train}\left[\text{wagon}3 \doteq Y_2 : \text{closed-car}\left[\text{load-set} \doteq \text{rectanglod}\right]\right]\\
& \vee\\
& X_3 : \text{train}\left[\text{wagon}3 \doteq Y_3 : \text{closed-car}\left[\text{load-set} \doteq \text{trianglod}\right]\right]\\
& \vee\\
& X_4 : \text{train}\left[\text{wagon}3 \doteq Y_3 : \text{open-car}\left[\text{load-set} \doteq \text{hexagonlod}\right]\right]
\end{aligned}$$

i.e. the *eastbound* trains are characterised by the `load-set` of the third wagon (which is the most discriminant feature), which may be a set of two `circlelod`, or a `rectanglod`, or a `trianglod` or a `hexagonlod`. Notice that implicitly this description requires that eastbound trains have at least three wagons (notice that the `load-set` feature in the obtained descriptions belongs to the `wagon3` feature). The descriptions obtained by DISC are similar to those obtained by the INDIE method (section 4.1.5 of chapter 5).

The *westbound* description obtained by DISC is also a disjunction of three feature terms:

$$\begin{aligned}
\text{west} = {} & X_1 : \text{train}\left[\text{wagon}2 \doteq Y_1 : \text{open-car}\left[\text{form-car} \doteq \text{openrect}\right]\right]\\
& \vee\\
& X_2 : \text{train}\left[\text{wagon}2 \doteq Y_2 : \text{closed-car-car}\left[\text{form-car} \doteq \text{jaggedtop}\right]\right]\\
& \vee\\
& X_3 : \text{train}\left[\text{wagon}2 \doteq Y_3 : \text{open-car}\left[\text{form-car} \doteq \text{ushaped}\right]\right]
\end{aligned}$$

i.e. the `form-car` feature is the more relevant to describe a west train. Note than these descriptions are also similar to those obtained by INDIE.

As we have explained in the section 4.1.5 of chapter 5, FOIL is not capable to obtain a description covering all the *westbound* trains and INDUCE uses constructive induction to introduce a new predicate in order to reach a complete description for the *westbound* class.

4.1.6. Traffic Law Dataset

This domain was introduced for testing the MOBAL system (Morik et al., 1993) and requires the use of background knowledge as was shown in INDIE (section 4.1.6 of chapter 5). Training examples are twelve traffic violation cases describing several violations such as *parking-violation*, *unsafe-vehicle-violation*, etc. Using some background knowledge rules, MOBAL is capable to obtain descriptions of concepts such as who is the responsible of a traffic violation, when a traffic violation is appealed, or when a traffic violation goes to the court.

DISC has been used to obtain descriptions for the same concepts as

INDIE, i.e. *parking-violation* and *lights-violation*. DISC has obtained the following description for *parking-violation* concept:

$$X_1 : \text{traffic-case}\left[\text{event} \doteq Y : \text{event}\left[\text{car-parked} \doteq \text{no-parking-place}\right]\right]$$

i.e. to parking a car in a `no-parking-place` constitutes a parking violation.

For, *lights-violation* concept DISC has obtained the following description:

$$X{:}\text{traffic-violation}\left[\text{event} \doteq Y{:}\text{event}\left[\text{involved-vehicle} \doteq Z{:}\text{vehicle}\left[\text{headlights-on} \doteq \text{false}\right]\right]\right]$$

i.e. a lights violation is committed when a car has not the headlights on. As in INDIE we have also asked for concepts as *court-citation* and *appeals*. DISC obtains for both concepts the same description as INDIE, namely:

$$X : \text{traffic-law}\left[\text{event} \doteq Y : \text{event}\begin{bmatrix}\text{involved-vehicle} \doteq Z : \text{vehicle}\begin{bmatrix}\text{owner} \doteq W : \text{person}\\ \text{sedan} \doteq \text{true}\end{bmatrix}\\ \text{responsible} \doteq W\end{bmatrix}\right]$$

4.1.7. Mesh Dataset

The Mesh dataset used by GOLEM (Dolsak and Muggleton, 1992) contains examples of three physical structures: a hydraulic press cylinder, a hook and a paper mill. These structures are represented qualitatively as finite collections of elements (meshes). Each structure is a set of several kinds of edges (e.g. important, not important, circuit, etc). Each edge can be related to other edges according to some geometrical constraints. The background knowledge in this dataset is described by types of edges, boundary conditions and loadings. DISC represent the objects of the Mesh dataset as feature terms belonging to the sort *edge*. Each edge can have from 5 to 11 features: `type`, `boundary-conditions`, `loadings`, `neighbour-XY-R`, `neighbour-YZ-R`, `neighbour -ZX-R`, `neighbour-XY-L`, `neighbour-ZX-L`, `neighbour-YZ-L`, `opposite-R` and `opposite-L`. See in appendix A the representation of the examples of the Mesh dataset. There are 17 solution classes to which the meshes can belong.

GOLEM obtains 56 different rules some which cover few examples and others that are not useful in practice. To reduce the final number of rules, GOLEM authors have eliminated some of the not useful rules and have allowed that rules cover some negative examples. LINUS (Lavrac and Dzeroski, 1994) and FOIL (Quinlan, 1990) have also been tested in this dataset, but they use five different structures (a hydraulic press cylinder, a pipe connector, a paper mill, a roller and a bearing box). We compare the results of DISC with those produced by GOLEM since we have not complete information (i.e. total number of rules, descriptions, etc) about the results of LINUS and FOIL.

As INDIE, DISC needs to control the depth of the feature terms since all objects of each structure are related and the use of complete feature term consumes a lot of time and memory. So, by taking as maximum depth 1, DISC has obtained 36 descriptions for class *one*, some of which are the following:

$$X_1 : \text{edge}\begin{bmatrix} \text{boundary-conditions} = \text{one-side-fixed} \\ \text{type} = \text{not-important} \end{bmatrix}$$

$$\vee$$

$$X_2 : \text{edge}\begin{bmatrix} \text{boundary-conditions} = \text{free} \\ \text{type} = \text{not-important} \end{bmatrix}$$

$$\vee$$

$$X_3 : \text{edge}\begin{bmatrix} \text{edge}\begin{bmatrix} \text{boundary-conditions} = \text{one-side-fixed} \\ \text{type} = \text{short} \end{bmatrix} \end{bmatrix}$$

In addition, there are 33 descriptions that, as in INDIE, correspond to the following patterns:

$$X_1 : \text{edge}\begin{bmatrix} \text{boundary-conditions} = \text{fixed} \\ \text{type} = \text{not-important} \\ \text{neighbour-ZX-L} \doteq W : \text{edge} \end{bmatrix}$$

$$\vee$$

$$X_2 : \text{edge}\begin{bmatrix} \text{type} = \text{short-for-hole} \\ \text{neighbour-ZX-L} \doteq S : \text{edge} \end{bmatrix}$$

$$\vee$$

$$X_3 : \text{edge}\begin{bmatrix} \text{boundary-conditions} = \text{free} \\ \text{type} = \text{short} \\ \text{opposite-R} = C5 \end{bmatrix}$$

$$\vee$$

$$X_4 : \text{edge}\begin{bmatrix} \text{boundary-conditions} = \text{fixed} \\ \text{type} = \text{short} \\ \text{loadings} = \text{cont-loaded} \\ \text{neighbour-ZX-L} \doteq P : \text{edge} \end{bmatrix}$$

$$\vee$$

$$X_5 : \text{edge} \left[\begin{array}{l} \text{boundary - conditions} = \text{fixed} \\ \text{type} = \text{short} \\ \text{loadings} = \text{not - loaded} \\ \text{neighbour - XY - L} \doteq \text{V} : \text{edge} \end{array} \right]$$

where the edges W, S, P and V take the value of a concrete edge. We have also used DISC to obtain the descriptions for class *two*. The result is 23 descriptions some of which are the following:

$$\begin{array}{l}
X_1 : \text{edge} \left[\text{type} = \text{long - for - hole} \right] \\
\vee \\
X_2 : \text{edge} \left[\begin{array}{l} \text{type} = \text{usual} \\ \text{neighbour - ZX - L} \doteq \text{S} : \text{edge} \end{array} \right] \\
\vee \\
X_3 : \text{edge} \left[\begin{array}{l} \text{type} = \text{short - for - hole} \\ \text{neighbour - ZX - L} \doteq \text{V} : \text{edge} \end{array} \right] \\
\vee \\
X_4 : \text{edge} \left[\begin{array}{l} \text{boundary - conditions} = \text{free} \\ \text{type} = \text{short} \\ \text{loadings} = \text{one - side - loaded} \end{array} \right] \\
\vee \\
X_5 : \text{edge} \left[\begin{array}{l} \text{boundary - conditions} = \text{free} \\ \text{type} = \text{short} \\ \text{loadings} = \text{not - loaded} \end{array} \right] \\
\vee \\
X_6 : \text{edge} \left[\begin{array}{l} \text{boundary - conditions} = \text{fixed} \\ \text{type} = \text{short} \\ \text{neighbour - ZX - L} \doteq \text{P} : \text{edge} \end{array} \right] \\
\vee \\
X_7 : \text{edge} \left[\begin{array}{l} \text{opposite - R} = \text{W} : \text{edge} \\ \text{type} = \text{not - important} \end{array} \right]
\end{array}$$

Where S, V, P and W are concrete edges such as B27, B36, etc. The description X_5 is also obtained by the GOLEM system, and there are other descriptions obtained by DISC that are equivalent to those obtained by GOLEM for class *one* (see appendix A).

4.2. Evaluation of the Accuracy

In this section we have evaluated the accuracy of DISC in order to assess its utility with respect to propositional learners. The evaluation of DISC has been made in the same domains as INDIE, (i.e. Lymphography and (small and Large) Soybean) datasets, and under the same conditions, i.e. attribute-value representation, training set containing the 67% of cases, and test set containing the remaining 33% of cases.

We have randomly constructed 20 training sets and the results of them have been averaged. As INDIE, DISC may answer multiple solutions and no solution. We have used the correctness function explained in section 4.2.1 of chapter 5 to analyse the information degree of the solutions with multiple answers.

DOMAIN	Method	Correct answer	No solution	Multiple solution	$G_1(E)$	$G_2(E)$
Soybean (small)	DISC	99,3%	1%	3%	0,981	0,976
Soybean (large)	DISC	88,7%	5,9%	8,1%	0,882	0,867
	CN2	81,6%	—	—	0,816	0,816
	C4.5	80%	—	—	0,800	0,800
Lymphography	DISC	72,4%	9,8%	5,1%	0,705	0,697
	CN2	81,7%	—	—	0,817	0,817
	C4.5	76,4%	—	—	0,764	0,764

Table 6.1. Results of the DISC method over Soybean (small and large) and Lymphography datasets. The results of CN2 and C4.5 are obtained from (Zhou and Dillon, 1995).

Table 6.1 shows results of DISC compared to those of the propositional learners CN2 and C4.5. Used over the large Soybean dataset, DISC has $G_1(E) = 0{,}882$ and $G_2(E) = 0{,}867$ of correctness. DISC produces a 8,1% of multiple solution answers and a 5,9% of no solution answers. The column in table 6.1 labelled as "correct answer" is the percentage of answers containing the correct solutions. Thus, if we consider that any answer containing the correct solution is a correct answer, then DISC provides an accuracy of the 88,7% over large Soybean, that is higher that the produced by the propositional learners.

Using DISC over the Lymphography dataset produces a $G_1(E) = 0{,}705$ and $G_2(E) = 0{,}697$ of correctness, with 9,8% of no solution answers and 5,1% of multiple solution answers. In column "correct answer" there is the percentage of accuracy considering that the multiple solution answers that contain the correct solution are correct.

An analysis of the answers having multiple solutions (see figure 6.5) shows that in both domains the highest number of solutions is three, although usually multiple answers have only two solutions. The correctness functions $G_1(E)$ and $G_2(E)$ over the large Soybean dataset show the higher correctness of DISC with respect to CN2 and C4.5. Instead, the correctness of DISC, used in the Lymphography dataset are lower than those of the propositional learners.

In order to see the DISC behaviour in dealing with domains with objects that have some features without unknown values, we have evaluated DISC over the small soybean domain. Both small and large Soybean have a common set of examples (all those examples in small Soybean are also present in large Soybean), but some of the examples in large Soybean are only partially described. Table 6.1 shows the results obtained by DISC over both Soybean datasets. Notice that the number of multiple answers is higher in the large Soybean dataset.

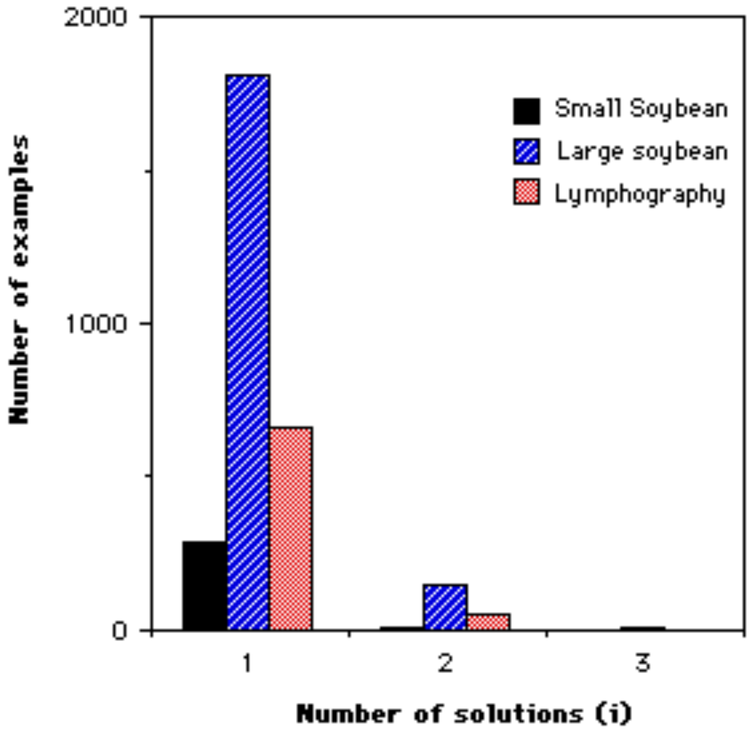


Figure 6.5. Analysis of the solutions with multiple answers produced by DISC over Soybean and Lymphography domains.

5. Conclusions

We have described DISC, a top-down method handling domain objects represented as feature terms. DISC searches for a discriminant description in the hypothesis space of feature terms using positive and negative examples. These examples can be partially described, i.e. some of their features can have unknown values.

Used over relational datasets, DISC has a behaviour similar to that exhibited by INDIE. The main difference between both methods is that DISC sometimes specialises eagerly and finds a larger disjunction of descriptions to characterise a class. The obtained descriptions are quite general (since they have few features) but the values of the features sometimes tend to be too specific. This is reflected in some of the relational domains, for instance in the combidrug description of the Drugs dataset.

Summarising, DISC has been capable to obtain descriptions in all the relational datasets in which it has been used, even in datasets such as trains or arch in which some relational learners have shown some problems. DISC tends to obtain class descriptions composed of many disjuncts, although each disjunct is a description having few features.

Concerning propositional datasets, the correctness functions $G_1(E)$ and $G_2(E)$ for the large Soybean dataset are higher for DISC than those of C4.5 and CN2 but lower than in the Lymphography domain.

The existence of domain objects having features with unknown values influences the DISC results by reducing the set of features that may be candidate to specialise the current description. When there are positive examples having some discriminant feature, say F, that is not common to all the examples of a class, F will not be a candidate to specialise the current description, and another feature F' will be selected. Nevertheless, the feature F may be candidate in the next specialisation. This situation occurs when F is common to the positive examples belonging to a set of the partition induced by F'.

Noisy values influence the results of DISC in a different way than INDIE. Let us suppose that A_i with a noisy value V is a feature candidate to specialise the current description. A_i only influences the DISC results if it is selected as the most discriminant. In such situation, the partition induced by A_i has at least, one more set (that corresponding to the value V). Since DISC builds at each step a description for each set of the partition, the final description may have one additional disjunct (that corresponding to the noisy value).

As future work we plan to improve DISC with a mechanism capable of dealing with imperfect data. A common criteria in most ILP systems for dealing with imperfect data is to relax the stopping condition, allowing, according to a threshold: (1) the covering of some predetermined number of negative examples, or (2) not covering some predetermined number of positive examples. In the future, the same criteria could be applied to DISC. Moreover, the bias used by DISC to determine the features that are candidates to specialise the current description may be relaxed. In particular, we want to study the DISC performance using a bias that takes as candidates any of the features belonging to the feature term obtained from the anti-unification of a (sub)set of examples.

Chapter 7

The LID Method

1. Introduction

The learning method proposed in this chapter, *Lazy Induction of Descriptions* (LID), is a lazy learning method for CBR applications. LID builds, in a lazy problem-centred way, a discriminant description for a specific problem. The basis of LID are the anti-unification operation and an entropy-reduction heuristic. Formally, LID is useful to solve the following task:

Given: - A set of precedents S classified in a partition of classes $C = \{C_1 \ldots C_n\}$
- A problem P to be solved

Find: A discriminant description D satisfying the following two conditions:

1) D subsumes P ($D \sqsubseteq P$)
2) For $S_D = \{s \in S \mid D \sqsubseteq s\}$, the set of precedents subsumed by D, all precedents in S_D belong to one class C_i.

These conditions are achieved by LID using a top-down strategy to build a description D subsuming both P and a subset of S (namely S_D). The top-down strategy uses an entropy-reduction heuristic over the set S_D. Entropy measures the disorder degree of a set with respect to a partition. In particular, a set whose elements belong to several classes has higher entropy than a set whose elements belong to less classes. LID uses the Shannon's entropy to estimate the disorder degree of the set S_D with respect to the partition C. The second condition of the task above is achieved when LID reaches a description D such that the corresponding set S_D has entropy zero. The description D finally obtained by LID can be interpreted as an explanation of why the problem P belongs to a class $C_i \in C$.

CBR methods usually retrieve precedents according to a set of indexes that have been determined during the Design phase (see section 3.2.2 in chapter 2). In other CBR systems, the similitude between cases is estimated using some metrics. Our proposal in designing LID is twofold. On the one hand we want to determine which are the relevant features in the retrieval process according to the problem to be solved whenever domain knowledge about feature relevance is not present. On the other hand, indexes and similitudes are used when domain objects are represented as attribute-value vectors. Because LID uses objects represented as feature terms, we propose an approach based on symbolic similitudes and heuristics allowing the assessment of the similitudes between feature terms.

In the next sections we provide first a general view of LID by means of its knowledge modelling analysis. Then, in section 3 we provide a detailed analysis of the LID algorithm. In section 4 we provide some results of the LID evaluation. Finally, in section 5 we explain a conversational version of LID. In chapter 9 can be found the application of LID over the marine sponges domain.

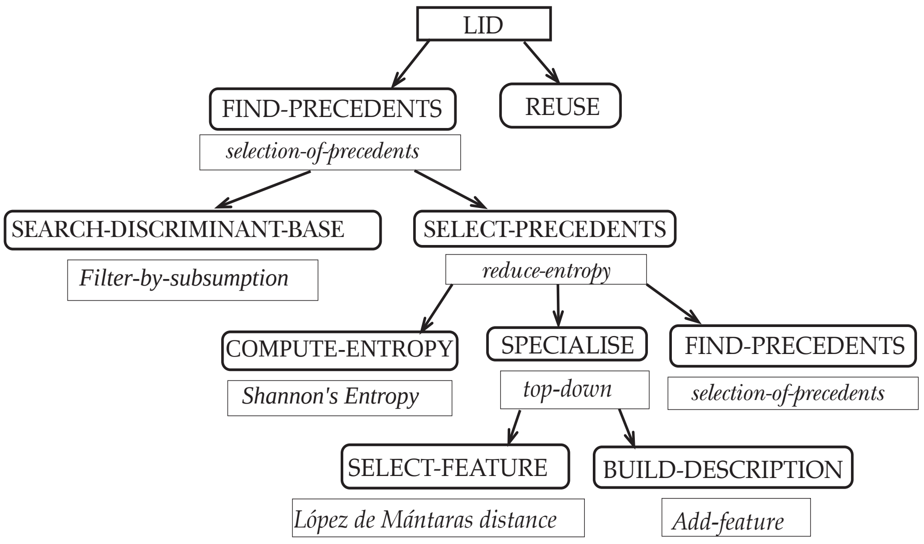


Figure 7.1. Decomposition of the LID method

2. General View of LID

Given a `description problem` model P, a `description` model D, a set of precedents S, and a partition of classes C = {C_1 ... C_n}, LID finds a subset $S_{D'} \subset S$ of solved episodes models such that all the problems described in $S_{D'}$ belong to the same class C_i. Moreover, LID builds a description D' that subsumes both the problem P and all the elements of the set $S_{D'}$.

The LID method decomposes in two tasks (figure 7.1): `find-precedents` and `reuse`. The `find-precedents` task has the goal of obtaining a set of precedents having entropy zero (or as close to zero as possible). From the set of precedents obtained by the `find-precedents` task, the `reuse` task elaborates a solution for P. The `find-precedents` task uses the `selection-of-precedents` method that decomposes in two tasks: `search-discriminant-base` and `select-precedents`.
Given a description D and the set of precedents S, the goal of the `search-discriminant-base` task is to obtain a subset of precedents $S_D \subset S$ subsumed by D. W e call this subset S_D the *discriminant base* with respect to the description D.

The `select-precedents` task has the goal of obtaining a discriminant base S_D having entropy zero. This goal is achieved by means of the `reduce-entropy` method that decomposes in three tasks: `compute-entropy`, `specialise` and `find-precedents`. The `compute-entropy` task uses the Shannon's entropy to estimate the entropy of the discriminant base S_D with respect to the partition C.

Let D be the current description, S_D the discriminant base associated to D and $H(S_D)$ the entropy of S_D with respect to the partition C. If $H(S_D)$ is not zero, the `specialisation` task uses a top-down method to specialise D. Intuitively, if D' is a specialisation of D and $S_{D'}$ is the discriminant base associated to D', then $S_{D'} \subset S_D$. The specialisation of D is made following the following steps (similar to those followed by DISC):

1) a feature term F that is the anti-unification of all the precedents in S_D and the problem P is obtained

2) the most discriminant feature in F is determined using the López de Mántaras distance. Let A_d the most discriminant feature.

3) D is specialised by adding the feature A_d with value the same value that A_d has in P. Let D' be the specialisation of D.

Once the description D' has been obtained by specialising D, the `find-precedents` task is used taking as input models the description problem P, the description D' and the discriminant base S_D. This process is repeated until either the entropy of the set of subsumed precedents is zero or it is not possible to reduce the entropy.

The `reuse` task of the LID method classifies the new problem according to the discriminant base S_D obtained by the `find-precedents` task. If the entropy of S_D is zero, all the elements in S_D belong to one unique class C_i; then the problem P can be classified as belonging to C_i. Nevertheless, if the entropy is not zero, several methods to solve the `reuse` task can be used. The easiest method is to produce a multiple solution answer. Thus, if the precedents in S_D can belong to several classes of a partition $C_D \subseteq C$, the `reuse` task can suggest that the new problem can belong to any class in C_D. In fact, this method can be improved producing a ranking of the classes C_D. A second method is to produce only one solution applying the *majority rule* to the classes C_D. In other words, the problem P can be classified as belonging to the class C_i to which the most of precedents in S_D belong.

In the next section we explain in detail the algorithm followed by the LID method.

3. Description of the LID Method

Given a set of precedents $S = \{e_1 \ldots e_m\}$ that can be classified in a partition of classes $C = \{C_1 \ldots C_n\}$, and a problem P to be solved, the goal of LID is to build a discriminant description D such that 1) D subsumes P, and 2) all the precedents subsumed by D belong to a unique class C_i. Each precedent e_i is a feature term and each feature A_i of a feature term has as values a set O_i where each $o_{ij} \in O_i$ is a feature term.

```
Let D = any
Function LID (P, S, D)
  $S_D$ := discriminant-base (S, D)
  H($S_D$) := entropy of $S_D$
  if H($S_D$) = 0 then reuse($S_D$)
             else D' := specialise (P, $S_D$, D)
                  LID (P, $S_D$, D')
  end-if
end-function
```

Figure 7.2. The LID algorithm.

The LID algorithm (see figure 7.2) follows a top-down strategy having three main steps. Given a set of precedents S and a description D (initially *any*), the first step uses the subsumption relation to obtain a subset of precedents S_D (the discriminant base) such that for all $e_i \in S_D$ we have $D \sqsubseteq e_i$. The second step is to estimate $H(S_D)$, the entropy of the discriminant base S_D. When the entropy of the discriminant base is zero then all the precedents contained in S_D belong to only one class C_i. In such situation, the third step of LID is to obtain a solution for P by means of the `reuse` task.

However, when the entropy of the discriminant base is not zero, the goal of LID is, according to the top-down strategy, to reduce the set S_D in order to reduce the entropy. In such situation, the third step of LID is to build a specialisation D' of the current description D with a smaller discriminant base $S_{D'}$. Next, LID recurs with parameters P, S_D and D', where P is the current problem to be solved, S_D is the subset of precedents subsumed by D, and D' is the specialisation of D. Notice that if D' is a specialisation of D, the respective discriminant base satisfies the relation $|S_{D'}| \leq |S_D|$. Thus, the goal of the specialisation operation is to obtain a subset $S_{D'}$ of precedents in S_D such that $H(S_{D'}) \leq H(S_D)$. This is a top-down strategy where descriptions are specialised in search of a description D such that $D \sqsubseteq P$ and the discriminant base S_D has entropy zero (or close to zero).

In the next sections we explain in detail how the entropy of the discriminant base is estimated, how the current description D is specialised, and how a solution for the problem P can be achieved.

3.1. Estimation of the Entropy

Given a partition of classes $C = \{C_1 \ldots C_n\}$ and the discriminant base S_D with respect to the current description D, LID uses Shannon's entropy to determine if the current description D is discriminant enough. The expression of the Shannon's entropy is the following:

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Where the function xlogx is defined as zero if x = 0. The entropy of the discrimination base S_D is estimated using the following expression:

$$H(S_D) = -\sum_{i=1}^{n} p_i \log_2 p_i \qquad \text{where } p_i = \frac{|S_D \cap C_i|}{|S_D|}$$

where C_i is a class in C and S_D is the discriminant base with respect to the description D.

The entropy measures the disorder degree of the set S_D with respect to the partition C. When all the elements in S_D are uniformly distributed among the sets in C, the set S_D has maximum entropy. Conversely, if all the precedents in S_D belong to a unique class, the entropy of S_D is zero. LID uses the entropy measure of the discriminant base S_D as an assessment of the discrimination degree of a description D.

```
Function SPECIALISE (P, $S_D$, D)
   $A_S$ := {Candidate features to specialise D}
   $a_d$ := select-feature ($A_S$, $S_D$)
   $v_d$ := value of $a_d$ in P
   D' := Add-feature (D, $a_d$, $v_d$)
end-function
```

Figure 7.3. Algorithm used to specialise the description D. The function add-feature(D, a_d, v_d) builds a description D' having the same features that D plus the feature a_d with value v_d.

3.2. Description Specialisation

Let D be the current description, S_D the discriminant base with respect to D and P the problem to be solved. The description D is specialised using a top-down strategy close to that used by the DISC method (section 3.3 in chapter 6). The first step (see algorithm in figure 7.3) is to decide which is the set of candidate features to specialise D. This set can be formed by any of the features appearing in the precedents. Nevertheless, as shown in DISC, since not all precedents use all features, we can use a bias that considers only the set of features A_c that are common to all the precedents and to the problem P. Let D_a be the feature term obtained by the anti-unification of P with all e_i, i.e. $e_i \sqcap P$ ($\forall e_i \in S_D$), i.e. all the precedents in S_D and the problem P. Let A_c be the set of features in D_a. The set A_c can be reduced by applying a second bias: to take only the subset A_S in A_c formed by the features that are leaves of the feature term D_a (notice that INDIE and DISC use the same biases).

```
Function SELECT-FEATURE ($A_S$, $S_D$)
   for each $a_i \in A_S$ do
        Let $P_{ai}$ be the partition induced by $a_i$ over $S_D$
        dist := {$d(P_{ai}, C_D)$ | López de Mántaras distance from $P_{ai}$ to $C_D$}
   end-for
   $d_{min}$ := min {$d(P_{ai}, C_D)$}
   return $a_d$ associated to $d_{min}$
end-function
```

Figure 7.4. Algorithm used to select a feature a_i to specialise the current description. C_D is the partition C containing only precedents in S_D.

The next step is to select one feature $a_d \in A_S$ to specialise the current description D and the algorithm is shown in figure 7.4. Let P_{ai} be the partition that a feature $a_i \in A_S$ induces on the set S_D. The goal is to select the most discriminant feature a_d using López de Mántaras distance that measures how similar are two partitions (see section 3.3 in chapter 5). LID uses López de Mántaras distance to estimate the distance from each partition P_{ai} to the correct partition C_D. The partition C_D is formed by the classes to which belong the precedents in S_D. The selected feature a_d is the feature inducing a partition P_{ai} with minimum López de Mántaras distance to the partition C_D[1].

Let v_d be the value that the feature a_d takes in P. The new description D' is the specialisation of D obtained by adding to D the feature a_d with value v_d. Notice that if $\{v_1 \dots v_n\}$ is the set of values that a_d takes in the precedents in the discriminant base S_D, it would be possible to obtain *n* specialisations of D (one for each value of a_d). Nevertheless, only D' satisfies D' $\sqsubseteq$ P. In other words, LID is interested only in building discriminant descriptions for one problem at time.

[1] Notice that this is different from the correct partition in INDIE and DISC. In these methods the correct partition has only two sets: one containing the examples belonging to one solution class C_k for which the description is constructed, and the other containing the remaining training examples.

3.3. Reusing Precedents

In this section we describe how the precedents in a discriminant base S_D can be reused in order to obtain a solution for P. The ideal situation holds when S_D has entropy zero since in such situation P can be easily classified from the precedents in S_D. However, there are some situations in which it is not possible to achieve a discriminant base with entropy zero. In this section we also analyse which are these situations and how LID can obtain a solution for P.

Let D be the current description and S_D the discriminant base with respect to D. If S_D has entropy zero then all the precedents in S_D belong to a unique class, say C_i. In such situation, the description D can be considered as a partial description of C_i, i.e. D is a correct description for a subset of elements in C_i (see next section). Because $D \sqsubseteq P$ also holds, LID can classify P into class C_i.

Let us suppose now that $H(S_D) \neq 0$ and D' is a specialisation of D such that $D' \sqsubseteq P$. In such conditions, the following two situations can occur:

1. The discriminant base $S_{D'}$ is empty. Let v_d the value that the most discriminant feature a_d takes in P, and D' the specialisation of D obtained by adding the feature a_d with value v_d. In such situation the discriminant base $S_{D'}$ is empty when there are not precedents in S_D taking the value v_d in the feature a_d.
2. Let A_s be the set of candidate features to specialise D. If there is no feature in A_s producing a specialisation D' of D such that $S_D \neq S_{D'}$, then the discriminant base cannot be reduced by specialising D.

Both situations above are abnormal stopping criteria in the sense that the solution for P cannot be directly obtained since D is not discriminant enough for P. The `reuse` task of LID has associated several methods, commonly used in CBR, that the user can choose to obtain a solution for P.

Let C_D the partition C restricted to the elements in S_D. If we are interested in a multiple solution answer, LID can use a method that produces as solution the set of classes in C_D. So, P can be classified as belonging to any of these classes.

A second method that also produces multiple solution answers is to rank the classes C_D according to the number of precedents of each class that also belongs to S_D. So, given two classes C_i and C_j such that $|C_i \cap S_D| = n_i$, $|C_j \cap S_D| = n_j$, and $n_i < n_j$ then the class C_i is a better solution than C_j. Moreover, the solution classes in C_D could be weighted, i.e. if $|C_i \cap S_D| = n_i$ then the class C_i is a solution for P with degree n_i.

Finally, LID can use a third method that applies the majority rule to the ranking produced by the second method above. Thus, LID proposes as solution for P the class C_i such that n_i is maximal.

3.4. Some Remarks

Given a problem P to be solved, let us suppose that D is a description having associated a discriminant base S_D such that $H(S_D) = 0$. Let C_i be the class to which all the precedents in S_D belong. In such situation, the description D can be considered as a partial description of C_i since it subsumes a subset of the examples in C_i.

Despite some similarities, LID has important differences with the methods INDIE and DISC. A main difference is that INDIE and DISC are eager learning methods whereas LID is a lazy learning method. This difference provides a different interpretation of the built descriptions. Thus, INDIE and DISC search for a description D for a given class C_i whereas LID wants to solve a new problem and during the problem solving process, a description is built. As a consequence, the description D built by INDIE and DISC characterises the class C_i, i.e. D is satisfied by *all* the examples in C_i. Instead, the description D built by LID is only satisfied by some of the examples in C_i, i.e. D is satisfied by the examples in C_i sharing some relevant features with the problem to be solved. In other words, the description D built by LID can be considered as the generalisation of the precedents $e_i \in C_i$ such that $e_i \in S_D$. Notice that the description D only can be associated to a class C_i when the discriminant base associated to D has entropy zero. LID is a lazy learning method since the description D that LID finds is used to solve a problem P and it is not used (nor intended to be used) to solve further problems.

4. Evaluation of the LID method

We have evaluated LID over some relational and propositional datasets. In particular, we have used both Robots and Mesh relational datasets and small Soybean, large Soybean and Lymphography propositional datasets. The evaluations have been made following the same conditions as used in DISC, and INDIE. We have constructed 20 training sets containing the 67% of the examples and the remaining 33% have been used as tests. These 20 training set are the same that those used in the evaluation of INDIE and DISC. In the next sections we analyse the results over each domain.

LID provides two kinds of answers: those having one (correct or incorrect) solution and those having multiple solutions. In chapter 9 the results of LIDin the domain of marine sponges identification can be found.

4.1. Robots Dataset

The Robots dataset (Lavrac and Dzeroski, 1994) consists of the descriptions of six robots (see section 4.1.1 in chapter 5). Two of them belong to the *friendly* class and the remaining four robots belong to the *unfriendly* class. Each robot is described by five features (see appendix A): `smiling`, `holding`, `has-tie`, `head-shape` and `body-shape`. Since LID is problem-centred, the evaluation of LID has to be made by identifying new robots. Because the robots dataset is small, we have defined the four new robots in Table 7.1.

	Attributes and Values				
Robot	Smiling	Holding	Has-tie	Head-shape	Body-shape
R8	yes	flag	yes	octagon	square
R9	no	balloon	yes	octagon	round
R10	no	balloon	no	round	octagon
R11	yes	flag	yes	square	square

Table 7.1. Description of new robots.

LID classifies the robot R8 as belonging to the *friendly* class obtaining the following description as justification of this membership:

$$D^8 = (\text{holding} = \text{flag}) \wedge (\text{head-shape} = \text{octagon})$$

By means of a detailed analysis of the steps followed by LID, we see that the more discriminant feature is `holding`. R8 has *flag* as value of this attribute, thus LID produces the following description:

$$D^8{}_1 = (\text{holding} = \text{flag})$$

The discriminant base associated to $D^8{}_1$ is the set {R2, R6}. Both robots have `holding` with value *flag*, but they belong to different classes, so the entropy of the discriminant base is 1. This means that $D^8{}_1$ has to be specialised using the next most discriminant feature. There are three features producing a discriminant base with entropy zero: `head-shape`, `has-tie` and `smiling`. LID has randomly chosen the `head-shape` entropy to specialise $D^8{}_1$. So, the result is the description D^8 above.

Nevertheless LID could have produced the following alternative descriptions if one of the other equally discriminant features were selected:

$$D^8{}_2 = (\text{holding} = \text{flag}) \wedge (\text{smiling} = \text{yes})$$
$$D^8{}_3 = (\text{holding} = \text{flag}) \wedge (\text{has-tie} = \text{yes})$$

Notice that descriptions $D^8{}_2$ and $D^8{}_3$ are the same obtained by DISC and LINUS (see section 4.1.1 in chapter 6).

The robot R9 is classified as belonging to the *friendly* class producing the following explanation:

$$D^9 = (\text{has-tie} = \text{yes}) \wedge (\text{head-shape} = \text{octagon})$$

Now, the most discriminant feature is `has-tie`, so the first description obtained by LID is the following:

$$D^9{}_1 = (\text{has-tie} = \text{yes})$$

that has the set {R1, R2, R3, R8} as discriminant base. Because the discriminant base contains examples belonging to both *friendly* and *unfriendly* classes, the description $D^9{}_1$ has to be specialised. The next most discriminant feature is `head-shape`, which is included in the specialisation of $D^9{}_1$ with value *octagon*. Thus, D^9 is a specialisation of $D^9{}_1$ having as discriminant base the set {R2, R8} that only contains robots belonging to the *friendly* class.

The robot R10 is classified as belonging to the *unfriendly* class producing as explanation the description:

$$D^{10} = (\text{has-tie} = \text{no})$$

that is one of the descriptions obtained by INDIE (section 4.1.1 in chapter 5).

Finally, the robot R11 is classified as belonging to the *friendly* class with the explanation:

$$D^{11} = (\text{has-tie} = \text{yes}) \wedge (\text{head-shape} = \text{square})$$

The first most discriminant feature is `has-tie`. When this feature takes as value *yes*, as R11, provides a description $D^{11}{}_1$ that has the set {R1, R2, R3, R8, R9} as discriminant base. This discriminant base contains examples of both solution classes, thus the obtained description $D^{11}{}_1$ has to be specialised. The feature selected to make this specialisation is `head-shape`. The description D^{11} is a specialisation of $D^{11}{}_1$ whose discriminant base is the set {R2, R8, R9} that contains only robots belonging to the *friendly* class.

Summarising, LID provides three descriptions for the *friendly* class and one description for the *unfriendly* class. These descriptions could be used to identify new robots before starting the complete LID process. This suggest a possible improvement of LID (see section 6).

4.2. Mesh Dataset

Domain objects of the MESH dataset are elements (meshes) composing three physical structures: a) hydraulic press cylinder, b) hook and c) paper mill. This domain has been used by GOLEM (Dolsak and Muggleton, 1992), FOIL (Quinlan, 1990) and LINUS (Lavrac and Dzeroski, 1994). However, FOIL and LINUS use five physical structures instead of three.

Objects in the Mesh dataset are represented in LID as feature terms belonging to the sort *mesh-problem* (see figure 7.5). Objects of this sort have two features: `description` and `solution`. Values of the `description` feature belong to the sort *edge*. Each *edge* feature term can have from 5 to 11 features:

`type`, `boundary-conditions`, `loadings`, `neighbour-XY-R`, `neighbour-YZ-R`, `neighbour-ZX-R`, `neighbour-XY-L`, `neighbour-ZX-L`, `neighbour-YZ-L`, `opposite-R` and `opposite-L` (see appendix A). The values of these features (except `type`, `boundary-conditions`, and `loadings`) are *edges* that have, in turn, some of the mentioned features. The `solution` feature of *mesh-problem* objects contains the class to which the described *edge* is classified. Meshes can be classified as belonging to 17 solution classes (form *one* to *seventeen*). For instance, the object in figure 7.5 belongs to class *one*.

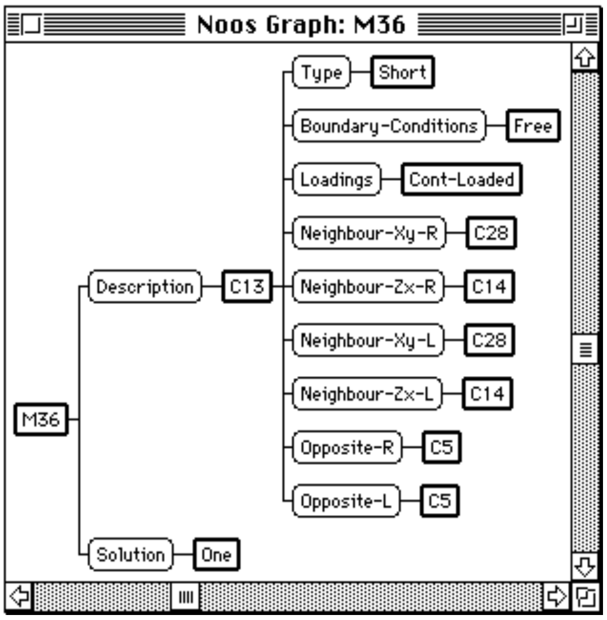


Figure 7.5. Representation of a Mesh domain object using feature terms.

We have used LID to identify the class to which a set of edges belongs. The conditions of this experiment have been the following:

- **Precedents**: - All the edges of the hook structure
  - All the edges of the paper mill structure
  - The edges of the hydraulic press cylinder belonging to the classes from *two* to *seventeen*.
- **Test Set:** Edges of the hydraulic press cylinder belonging to the class *one*.

Edges in the test set are not classified, that is to say, LID does not know the class to which they belong. So, LID has classified the 21 edges of the test set as follows:

- 7 edges have been correctly classified as belonging to the class *one*
- 8 edges have been incorrectly classified as belonging to the class *two*
- 6 edges have been classified (after applying the majority rule in `reuse` task) as belonging to either classes *one* or *two*

Notice that edges could have been classified as belonging to any of the 17 solution classes, but LID produces answers having as maximum two solutions (classes *one* and *two*).

We only have accuracy results of FOIL, mFOIL and GOLEM used over five physical structures (hydraulic press cylinder, hook, paper mill, roller, and bearing box). The evaluation of these systems has been made using a leave-one-out strategy, i.e. all the edges of a structure have been identified one by one using the edges of the remaining structures (Lavrac and Dzeroski, 1994). Under these conditions, the accuracy of FOIL, mFOIL and GOLEM is, respectively, 12%, 21% and 19%.

These results are not comparable to those obtained using LID since results of FOIL, mFOIL and GOLEM are the mean value of the accuracy obtained from the identification of all the edges belonging to one physical structure. Results of LID are obtained from the identification of only a subset of edges of one physical structure.

For each edge, LID has also produced a description D that can be viewed as an explanation of the classification, when the entropy of the discriminant base S_D is zero. We are interested in analysing these descriptions in order to improve LID (see section 6). In particular, for the 7 edges classified as belonging to class *one*, LID has produced the following explanation:

$$X_1 : \text{edge}\begin{bmatrix}\text{loadings} = \text{not-loaded}\\ \text{type} = \text{not-important}\end{bmatrix}$$

that is the same description obtained by GOLEM for class *one*.

LID has incorrectly classified 8 edges as belonging to class *two*. However, the descriptions provided by LID as explanation of the classification have been the following:

$$D_1 = X_1 : \text{edge}\begin{bmatrix}\text{loadings} = \text{not-loaded}\\ \text{type} = \text{not-important}\\ \text{loadings} = \text{cont-loaded}\end{bmatrix}$$

$$D_2 = X_2 : \begin{bmatrix}\text{loadings} = \text{one-side-loaded}\\ \text{type} = \text{short}\end{bmatrix}$$

D_1 is not comparable to the descriptions obtained by the other methods (GOLEM, FOIL, INDIE and DISC) and D_2 is similar but more general than X_4 obtained by DISC (see section 4.1.7 in chapter 6). Nevertheless, both descriptions D_1 and D_2 only subsume edges belonging to class *two*.

We have made a second experiment changing the set of edges to be identified and, consequently, the set of precedents. In particular, the conditions of this second experiment have been the following:

- **Precedents**: - All the edges of the hook structure
  - All the edges of the paper mill structure
  - The edges of the hydraulic press cylinder belonging to classes *one* and *three* to *seventeen.*
- **Test Set:** Edges of the hydraulic press cylinder belonging to class *two.*

Now, the test set contains 15 unclassified edges that LID has identified in the following way:

- 2 edges have been correctly classified as belonging to class *two*
- 9 edges have been incorrectly classified as belonging to class *one*
- 4 edges have been classified (after applying the majority rule in the `reuse` task) as belonging to several classes.

As in the previous experiment, we want to analyse the descriptions obtained by LID. So, the explanation given by LID of why an example belongs to class *two* is the following:

$$X_1 : \text{edge}\left[\text{type} = \text{long - for - hole}\right]$$

that is the same description obtained by DISC (section 4.1.7 in chapter 6).

The explanations of why the 9 incorrectly classified examples have been classified as belonging to class *one* are the following:

$$D_1 = X_1 : \text{ edge }\begin{bmatrix}\text{loadings} = \text{not - loaded}\\ \text{type} = \text{short - for - hole}\end{bmatrix}$$

$$D_2 = X_2 : \text{ edge }\begin{bmatrix}\text{type} = \text{short - for - hole}\\ \text{neighbour - ZX - L} \doteq V : \text{ edge}\end{bmatrix}$$

$$D_3 = X_3 : \text{ edge }\left[\text{type} = \text{not - important}\right]$$

$$D_4 = X_4 : \text{ edge }\begin{bmatrix}\text{type} = \text{short}\\ \text{loadings} = \text{one - side - loaded}\end{bmatrix}$$

$$D_5 = X_5 : \text{edge} \begin{bmatrix} \text{type} \ = \ \text{short} \\ \text{loadings} \ = \ \text{no - loaded} \\ \text{boundary - conditions} \ = \ \text{fixed} \end{bmatrix}$$

where V can take as valueA24 or A25.These descriptions are not comparable with those obtained by the methods FOIL, GOLEM, INDIE and DISC.

As conclusion we want to emphasise the LID performance over the Mesh dataset since the obtained descriptions seem more general and accurate than those obtained by INDIE and DISC.

4.3. Small Soybean Dataset

This dataset contains 49 examples belonging to four solution classes: *diaporthe-stem-canker*, *charcoal-rot*, *phytophtora-rot* and *rhizoctonia-root-rot*. Each one of these solution classes has 10 examples, except *phytophtora-rot* that has 19 examples. All the examples are completely described, i.e. there are not features having unknown values (see appendix A).

Method	Correct answers	No solution	Multiple solutions	$G_1(E)$	$G_2(E)$
LID	100%	0	0	1	1
INDIE	99%	1%	4%	0,977	0,970
DISC	99,3%	1%	3%	0,981	0,977

Table 7.2. Results of the LID method applied to small Soybean Database.

Table 7.2 shows the results of the LID method compared with those obtained by INDIE and DISC. LID improves the results produced by both methods. In particular DISC has a good behaviour in this domain but some examples are not classified and also has multiple solution answers. The value of the correctness functions using LID is 1 (i.e. LID has a 100% of accuracy), no multiple solutions and no problem is left unclassified.

Mainly, the higher correctness of LID is due to small Soybean dataset is very regular in the sense that there are not training examples having features with unknown values. However, notice that in this "perfect" dataset LID obtains "perfect" results while other methods do not.

4.4. Large Soybean Dataset

This dataset is composed by 306 examples, 49 of which are also present in the small Soybean database. Examples have partial information, i.e. they may have features with unknown values. There are 19 solution classes (see appendix A) containing as a mean 10 training examples (but there is one class containing 40 examples whereas others contain 1 or 3 examples).

Method	Correct answers	$G_1(E)$	$G_2(E)$
LID	80,8%	0,808	0,808
CN2	81,6%	0,816	0,816
C4.5	80%	0.800	0,800

Table 7.3. Results of the LID method applied to large Soybean Database.

The result of the LID application over the large Soybean dataset is shown in table 7.3. LID does not produce multiple solution answers and always provides a solution. Correctness functions of LID are comparable to correctness functions of CN2 and C4.5. In fact, because LID always produces at least one answer, the correctness functions are equivalent to the accuracy measure (see section 4.2 in chapter 5).

The large Soybean dataset has features with unknown values. Notice that unknown values in INDIE and DISC yield overgeneralised descriptions that subsume examples belonging to several classes. As a consequence, both methods produce multiple solution answers. Unknown values in LID also produce more general descriptions that, in turn, have associated discriminant bases with higher entropy. Nevertheless, several methods can be used in LID to avoid multiple solutions. In particular, we have used the majority rule method.

We have observed that the majority rule can increase the number of incorrect answers when the number of examples in each solution class is very different. Let us suppose that in solving a problem P, LID has built a description D with a discriminant base S_D. Let us also suppose that S_D contains examples belonging to two solution classes C_c and C_i, where C_c is the correct solution for P and C_i is an incorrect solution. In such situation, if $C'_c = \{e_j \mid e_j \in C_c \cap S_D\}$, $C'_i = \{e_j \mid e_j \in C_i \cap S_D\}$, and $\mathrm{Card}(C'_c) < \mathrm{Card}(C'_i)$, LID will provide C_i as the solution class for P. In other words, LID provides a unique solution that is incorrect whereas without applying the majority rule LID provides a multiple solution answer that includes the correct solution. As a consequence, the utility of using the majority rule has to be determined for each domain. This issue also appears when LID is used to identify marine sponges (see chapter 9).

4.5. Lymphography Dataset

This dataset is composed of 148 examples that can belong to four solution classes: *malign-lymph*, *metastases*, *fibrosis* and *normal-find*. Examples in this domain are described by 18 features none of which has unknown values. Solution classes of the Lymphography dataset are composed of a very different number of examples. So, the *malign-lymph* class contains 61 examples; the *metastases* class contains 81 examples; the *fibrosis* class contains 4 examples; and, finally, the *normal-find* class contains 2 examples.

As in the Soybean datasets, the LID method has not produced multiple solution answers nor no solution answers.

Method	Correct answers	$G_1(E)$	$G_2(E)$
LID	74,4%	0,744	0,744
CN2	81,7%	0.817	0,817
C4.5	76,4%	0,764	0,764

Table 7.4. Results of the LID method applied to Lymphography Dataset.

Table 7.4 shows the results of the application of LID over the Lymphography dataset. In this table we show that the correctness functions of LID are lower but comparable to those of the C4.5 method. However, it is not possible to state whether the difference among LID and the propositional learners is significant or not since we have not used the same set of training examples than those used by C4.5 and CN2 in the results reported.

5. Conversational LID

Aha and Breslow (1997) defined *Conversational Case-based Reasoning* (CCBR) as a CBR process that iteratively interacts with a user in a conversation in order to solve a task. The goal of the CCBR is to retrieve precedents making as few questions as possible to the user.

Aha and Breslow use a top-down algorithm to induce a decision tree from cases in order to make the minimum number of questions. The construction of a decision tree implies the selection of the most discriminant attributes. Usually, this selection is made taking into account that all the examples from which the decision tree has to be induced have the same attributes. However, cases can have few attributes in common, so this selection criterion has to be changed. In particular, Aha and Breslow propose to select the most frequently used attribute in order to partition the case base.

```
Initialisation: D = any
Function CLID (S, D)
  $S_D$ := discriminant-base (S, D)
  $H(S_D)$ := entropy of $S_D$
  if $H(S_D)$ = 0 then reuse($S_D$)
              else D':= conversational-specialisation ($S_D$, D)
                   CLID ($S_D$, D')
  end-if
end-function
```

Figure 7.6. The CLID algorithm.

The LID algorithm (explained in section 3) can be used for CCBR with some modifications. We call CLID the conversational version of LID. Given a set of precedents S = $\{e_1 \dots e_m\}$ that can be classified in a partition of classes C = $\{C_1 \dots C_n\}$, and a problem P to be solved, the goal of CLID is to build a discriminant description D such that 1) D subsumes P, and 2) all the precedents subsumed by D belong to a unique class C_i. That is to say, CLID has the same goal than LID. The strategy followed in CLID is also the same than that of LID (compare algorithms in figures 7.2 and 7.6): the main difference between both algorithms is that the problem P to be solved is a parameter of LID but it is not a parameter of CLID. In other words, LID initially knows all the information about P and uses it during the construction of D. Instead, the only information that CLID knows about P is that asked to the user.

```
Function CONVERSATIONAL-SPECIALISATION ($S_D$, D)
   $D_a$ := anti-unification ($S_d$)
   $A_S$ := {leaves of $D_a$}
   $a_d$ := select-feature ($A_S$, $S_D$)
   c := path from root($D_a$) to $a_d$
   $A_c$ := features in c whose value has not been asked
   Ask-path-to-user ($A_c$)
   while all features in $A_c$ have unknown value do
        $a_d$ := select-next-feature ($A_S$, $S_D$)
   end-while
   D' := Add-feature (D, c)
end-function
```

Figure 7.7. Algorithm used by CLID to specialise the description D. `Conversational-specialisation` task ask to the user for the value that the selected feature takes in the problem P to be solved.

The conversational part of CLID is included in the `conversational-specialisation` task (see figure 7.7). Let D be the current description and S_D the discriminant base of D. The bias of CLID takes as feature candidates to specialise D only the subset A_S formed by the features that are leaves of the feature term D_a obtained from the anti-unification of the precedents in S_D. Notice that in LID the feature term D_a was obtained from the anti-unification of both the precedents in S_D and the problem P.

As before, the most discriminant feature a_d is selected using López de Mántaras distance for all features in A_s. Then, CLID asks the user the value of a_d in the current problem. Notice that the selected feature a_d is a leaf of the feature term D_a. This means that there is a path *c* from the root of D_a to a_d. Notice that some features (for instance the root) of the path *c* can belong to several paths. Therefore, the path *c* can include features that have already been asked as a consequence of the previous selection of other leaf features. Let A_c be the set of features whose values have not yet been asked. If the user answers that he does not know any value for any of the features in A_c then the next most discriminant feature has to be selected and asked. Otherwise (i.e. the user answers some value for a feature in A_c), the path, although incomplete, has to be included to the specialisation of D.

The main difference between CLID and LID is that CLID does not use the current problem in the anti-unification operation for selecting candidate features. A consequence is that the feature selected as the most discriminant may be unknown in P. This situation does not appear in LID since the most discriminant feature is obtained from the anti-unification of both, the precedents and P.

6. Conclusions

The LID results are quite similar to that exhibited by the DISC method. In fact, LID often builds a description for a solution class C_k that is one of the disjuncts of the description for C_k obtained by DISC. Inductive learning methods such as INDIE or DISC build descriptions that subsume all the examples of a class, while the descriptions built by LID are a discriminant description valid for a subset of examples of a class.

Differently than INDIE and DISC, LID does not provide multiple solution answers, since it applies the majority rule. Nevertheless, the majority rule can increase the number of incorrect answers, specially when the solution classes of a domain have a very different number of examples (as explained above in the large Soybean dataset).

From a CBR point of view, LID is interesting as a method allowing the retrieval of past cases with a structured representation without using domain-specific knowledge to determine which features are relevant for a task. The description D built by LID can be considered as the set of relevant indexes allowing the retrieval of precedents relevant to the current problem. Let P_i be a problem to be solved and D_i the discriminant description having a discriminant base S_{Di} such that $H(S_{Di}) = 0$.

LID is a lazy method since it does not store the built descriptions. As future work we will consider to store each built description in order to be used to solve new problems. Let us suppose that LID stores all the obtained descriptions D_i, and we call this set *I*. Given a problem P to be solved, LID could search in the set *I* for a description D_i such that $D_i \sqsubseteq P$. If such D_i is found, the problem P can be identified as belonging to the class C_i to which all the precedents in S_{Di} belong. Otherwise, the pure lazy version of LID (the one explained in this chapter) has to be applied.

Notice that if each generated description D_i is stored and used to solve new problems, the LID method can be seen as a method between eager and lazy learning methods. The storage of the description implies the analysis of some issues such as how the stored descriptions have to be updated with the resolution of new problems or how to detect incompatibilities between descriptions of the same class.

Conclusions Part II

Feature terms formalism is a subset of first-order logic. Most of relational datasets have been studied in the context of ILP, using Horn clauses (that is a subset of first-order logic) as representation formalism. We have shown in Part II how several strategies for relational learning can be designed and implemented as *learning methods* for feature terms. Two of them are inductive learning methods: INDIE and DISC, and the third one, LID, is a lazy learning method.

The anti-unification concept is used in INDIE, DISC and LID, although each method uses this concept in a different way. INDIE uses the anti-unification as a basis to search a class description whereas DISC and LID use the anti-unification as a bias to restrict the set of features that are candidates to specialise a class description.

INDIE, DISC and LID have been applied over relational and propositional datasets. Applied over relational domains, these methods have produced results similar to those of other relational learners such as GOLEM, LINUS and FOIL. In particular, the proposed methods have reached a solution in all the domains in which they have been applied, whereas some of the relational learners cannot. For instance, FOIL cannot found a description for the *westbound* class of the trains dataset, and also, in the Arch dataset FOIL finds a escription that is not correct. In some datasets, INDIE and DISC build descriptions composed of may disjuncts (for instance, the descriptions of the Mesh domain). This issue is a consequence of the unknown values and it is discussed later.

Applied over propositional domains, our methods have produced results comparable to those of propositional learners such as C4.5 and CN2. When INDIE and DISC have been applied to propositional domains, they may provide multiple solutions answers or no solution answers. We have proposed a correctness function allowing the estimation of the goodness of the method. Two approaches: $G_1(E)$ and $G_2(E)$ have been used as correctness functions.

The anti-unification of partially described examples produces a description that is more general than the description obtained when all the examples are completely described. Such situation is due to both the representation of unknown values using feature terms and the use of the anti-unification operation as bias. When a feature F of an example E has unknown value, F does not appear in the description of E. Moreover, F will not appear in the description D obtained anti-unifying E with other examples, since D contains only the features common to all the anti-unified examples.

INDIE anti-unifies the positive examples of a class C to build a description for C. Therefore, when there are examples with unknown values, INDIE tends to produce more general descriptions. In turn, more general descriptions tend to produce more multiple solution answers.

DISC and LID use anti-unification as bias to determine a set S of features candidate to specialise a currently over-generalised description D. In both methods, the set S contains the features of the feature term obtained from the anti-unification of the examples. Thus, partially described examples reduce the set S, since only features common to all the examples are candidates. This reduction, in turn, can affect the result of the heuristic used to search the most discriminant feature in order to specialise the current description D. That is to say, if some discriminant feature is not contained in the set S the heuristic can never select it. In such situation, DISC tends to obtain a final description with a larger number of disjuncts. Concerning LID, the discriminant base associated to the obtained description tends to have a entropy higher than zero.

On the other hand, features with noisy values influence the results of the proposed methods in two ways: 1) tending to produce unnecessary general descriptions (as in the INDIE method), 2) tending to produce a disjunction having a larger number of disjuncts (as in DISC). Concerning LID, noisy values affect the results just as unknown values: they tend to produce more general descriptions with discriminant bases having higher entropy.

Summarising, we were specially interested to analyse the use of feature terms in relational domains. For this reason, the experiments with INDIE, DISC and LID have mainly been made over relational datasets whereas only two propositional datasets have been used. The results of induction using DISC and INDIE have been comparable to those produced by relational learners such as LINUS, FOIL and GOLEM. LID also works adequately but results are not comparable since it is not a pure inductive learning method.

Concerning propositional datasets, the INDIE and DISC performance is comparable to that of C4.5 and CN2 (as before, the LID results are not directly comparable since it is a lazy learning method).

There are several issues (such as numerical values and noise) that have not been addressed in dealing with propositional learners. Both questions will be addressed in the future. On the other hand, the use of feature terms allows the representation of objects partially described, i.e. having features with unknown values.

As future work we plan to improve the INDIE, DISC and LID methods with a mechanism capable to deal with imperfect data. Both noisy values and unknown values could be handled in two ways: 1) allowing the construction of descriptions subsuming some negative examples (as in GOLEM and CN2), or 2) allowing the construction of descriptions that do not subsume all the positive examples (as in FOIL). In fact, both threshold mechanisms can be incorporated together to our methods.

Commonly, propositional learners use objects described by a fixed vector of attribute-value pairs. This representation presents some problems in dealing with objects having attributes whose values depend on the value of other attributes. For instance, let us suppose that the attribute A_2 only makes sense if the attribute A_1 has the value v. In such situation, when A_1 has a value different to v, the value assigned to A_2 is DNA (*does not appear*). This situation in structured representation is easily modelled by defining the value of A_1 as a structured object that has A_2 as attribute. That is to say, A_2 is an attribute of a subcomponent of the problem. The translation from the vector of attribut-value pairs into a structured representation requires domain information for establishing which subcomponents are to be used. We do not have the domain knowledge required to translate the standard datasets, in a meaningful way, to a structured representation. Thus, in fact, we have not used the power provided by the feature terms representation, since we have represented vectors by means of feature terms.

As future work we want to address the questions above mentioned (numerical values, noise and structured representation of datasets). Once these questions have been addressed we will experiment with other propositional datasets.

The main contributions of the part II are the following:

- Definition of methods that use feature terms to represent both relational and propositional datasets.
- Definition of a heuristic bottom-up inductive learning method (INDIE).
- Definition of a heuristic top-down inductive learning method (DISC).
- Definition of a lazy learning method (LID) that does CBR retrieval for structured representation of cases.
- We have proposed a correctness function to estimate the goodness of methods when they can provide answers with multiple solutions. The accuracy is a particular case of this correctness function.

PART III

MOTIVATION

In this part we have developed some prototype applications that show the feasibility of the framework proposed in chapter 2 and the methods of the part II. Mainly we want to show that the proposed integration of problem solving and learning. Secondly, we want also show the utility of the lazy problem-centred approach to search for appropriate methods to solve the tasks.

Thus, in chapter 8 we describe CHROMA, an application supporting the search for an appropriate plan to purify a protein. CHROMA shows integration of problem solving and learning and also lazy problem-centred selection of the appropriate method for a task.

In chapter 9 we describe SPIN, an application to identify marine sponges. This application integrates several learning methods and problem solving. In particular, SPIN uses the learning methods INDIE, DISC and LID explained in the Part II. Objects of this domain are sponges, and they are described in a structured representation using feature terms.

Chapter 8

CHROMA: A Support Tool for Protein Purification

1. Introduction

When bioscientists are interested in the analysis of the behaviour of a molecule they should obtain this molecule in a pure form from a biological sample. This sample could be a biological tissue (animal or vegetal) that contains, besides the molecule of interest, hundreds of other molecules that should be removed. Protein purification is the process that allows the isolation of one molecule among many others. Once the molecule of interest has been isolated, its structure, function, electrical and physical properties and behaviour can be analyzed. The development of techniques and methods for the separation and purification of biological macro-molecules (such as proteins) has been an important prerequisite for many of the advancements made in biosciences and biotechnology over the past three decades. The main problems that can appear in a purification process are in general related with denaturation, proteolysis and contamination with pyrogens, nucleic acids, bacteria and viruses. These problems can be limited by the proper choice of an extraction medium.

The purification process may be analytical or preparative. An analytical purification is made when there is a small amount of the molecule to purify. This kind of purification is used to adjust an appropriate purification process. A preparative purification process has as goal the purification of a greater amount of a molecule.

A purification process is composed by one or several steps that can be included in three groups: fractionation and extraction, electrophoresis and chromatographic techniques. Extraction and fractionation steps are usually based on precipitation and centrifugation techniques, and they are useful to make a first cleaning of the sample. Electrophoresis (from Greek *elektros:*

electricity and *foresis: movement*) allows to separate molecules in solution according to their net electric charge or the ratio mass/charge. Electrophoresis is a powerful analytical tool, widely used in the biochemical laboratory.

The term *chromatography* groups a set of separation techniques based on the movement of the molecules between two phases: one stationary phase and the other mobile phase. The chromatographic techniques allow the obtention of purified molecules using analytical or preparative processes. More information about the chromatographic techniques can be found in appendix B.

The first step to purify a molecule is to choose a source (sample) containing a sufficient protein concentration. This source has to be easily available, and the protein of interest into this source has to be stable enough, possible interferences and stranger activities have to be known and controllable. The second step is to design a purification scheme useful to obtain the protein of interest sufficiently pure to make the analysis we want.

There are two ways to proceed. The first one is to take an available source. It is advisable to use as source materials constituting a proteinic solution not very complex (for example, blood, urine, snake venom or extra-cellular mediums from cultures) since that they contain a limited number of non-desired components. In contrast, a sample from a tissue has a lot of contaminants and the adjustment of the process may be difficult. Once the source has been chosen, it has to design and adjust, according to the characteristics of the different chromatographic techniques (described in previous sections), a purification scheme achieving the expected result. This way is not easy and the adjustment of a purification process can take months.

The usual way is to look in the literature for previos purifications of the protein that we are interested in. Then we can use the same source than the obtained experiments and, consequently, the same purification process will be useful. Let us suppose that we have found in the literature an experiment purifying the Alkaline Phosphatase protein from dog liver using three steps. If we are also interested in the Alkaline Phosphatase but our source is fish, we can assume that the same purification process with a few variations may be useful[1] .

The main difficulty of this way is the unavailability of the sources used in the obtained literature. Therefore, the purification process has to be modified according to the characteristics of the available source. The optimisation of the chosen purification process is made by a systematic variation of parameters as the composition of the extraction method. The extraction of a protein from a solid source implies an agreement between the retrieval of the protein and its purity.

[1] In fact, this is a very usual assumption due to few available information about the variability of the proteins in the different species. But there are proteins having different physico-chemical features according to the species from which they are obtained.

There is a commercial system called FPLCassistant™ sponsored by Pharmacia (Osterlund, 1993) that also supports the process of search for a purification plan. Its knowledge base is composed of text book knowledge in form of a database to which the user can access. The user gives the parameters of its particular purification and the system recommends which techniques may be used and which not. Once the user has chosen a technique, he can introduce parameters such as volume, pressure, etc and the system computes the condition under which the chosen technique can be applied.

CHROMA has a base of cases containing experiments obtained from the literature (*Comparative Biochemistry and Physiology* revue). CHROMA searches in this base and the result of this search is one or several experiments close to our experiment providing a first approximation of how the protein of interest can be purified. We want to make special emphasis in that the adequacy of the proposed solution can be only evaluated in the laboratory. That makes difficult the evaluation of CHROMA.

In the next sections we present the CHROMA application, detailing its knowledge modelling, its implementation and its evaluation.

2. Description of CHROMA

Given a protein to purify, the main goal of CHROMA is to obtain a plan purifying the protein. This goal is achieved by the `purification` task. From a knowledge engineering phase we have detected that searching past purification experiments is an essential part in the human expert solving the purification task. We have modelled this as a CBR system, called CHROMA. CHROMA is based on purification cases, and it is capable to find precedent cases useful for solving new experiments. A requirement that emerged in the knowledge engineering phase was that the user of the system has to have the final decision about which plan is finally chosen. This requirement arises from the fact that the user is knowledgeable of the chemical domain and wants to maintain control on the purification process. The CHROMA application supports the user in inspecting the candidate cases proposed for taking his final decision.

In this section we describe the models used in the CHROMA application. Then we show how the `purification` task is solved. The implementation of CHROMA is made using the NOOS language described in chapter 3.

2.1. Models in CHROMA

Domain knowledge used in CHROMA is constituted by 108 purification experiments obtained from the *Comparative Biochemistry and Physiology* revue. Each purification experiment contains the sample from which a protein has been purified and the purification plan that has been used. Experiments in CHROMA are represented by feature terms belonging to the sort *experiments* having two features: `description` and `purification` (see figure 8.1). `Description` is a feature whose value is an object belonging to the *description-experiment* sort. Objects in the *description-experiment* sort have two features: the `protein` purified in the experiment, and the `sample`. A `sample` is described in turn by two features: the `species` where the sample comes from, and the `source` of the sample (an animal or vegetal tissue, a culture, etc).

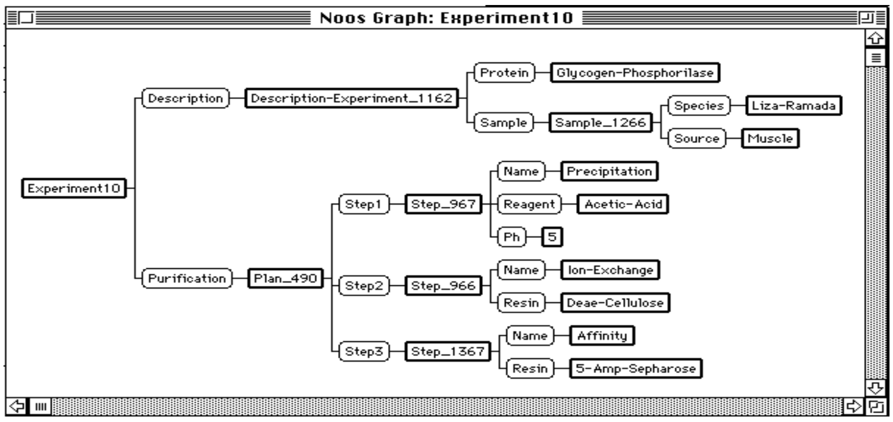


Figure. 8.1. Description of an experiment from the case-base of the CHROMA application. An experiment is composed of a description, containing the protein to purify; a sample from which the protein has to be purified; and the purification plan composed of several steps.

The `purification` feature has as value a feature term belonging to the *plan* sort having a feature (*steps*) for each chromatographic step. For example, the experiment in figure 8.1 uses a purification plan having three features: `step1`, `step2` and `step3`. Each `step` has the `name` of the chromatographic technique (affinity, gel filtration, etc.) and the name of the substance (`reagent` or `resin`) used to purify the protein[2]. For instance, the purification plan of *experiment10* shown in figure 8.1, uses as first step an Acetic Acid Precipitation, as second step a Ion Exchange with DEAE-Cellulose and as third step an Affinity with 5-AMP-Sepharose.

A new experiment to purify has only the `description` feature and the `purification` task has as goal to complete the `Solved Episode` model (see section 2.1.1 in chapter 3), i.e. to find a value for the `purification` feature .

[2] Eventually, a step can have additional features such as the pH or the pI.

2.2. Solving the Purification task

Given a new experiment and a base of solved experiments, the goal of the `purification` task is to find a sequence of chromatographic techniques (purification plan) purifying the protein of the new experiment. The domain expert uses different strategies to find a purification plan for the current protein:

M1) Searching for an experiment using exactly the same sample for the same protein.

M2) Searching for experiments purifying the same protein but from other kinds of sample. If more than one is found, the domain expert chooses one of them according to some specific criteria.

M3) If the sample of the current experiment satisfies some specific domain properties (i.e. the current protein belong to a special family of proteins), the domain expert knows which purification plan to apply without searching for past experiments.

M4) If the domain expert has not found any experiment in the literature purifying the protein of the current experiment, he tries to build a purification plan by trial and error in the laboratory. The steps of this purification plan are build according to the characteristics of each purification techniques.

Each of these strategies has been modelled in CHROMA by a different problem solving method. In particular, strategy M1 has been modelled by the `equal-sample` method that detects if there is an experiment in the case base having the same protein and sample as the current experiment. The `analogy-by-determination` method is a case-based method, used to model strategy M2, that retrieves experiments from the case base that purify the same protein. Given a protein P, several experiments purifying P can be retrieved: the `analogy-by-determination` method performs some interaction with the user in order to let him decide the most appropriate precedent.

Strategy M3 has been modelled by a classification method called `purify-by-class`. This method uses intensional concept descriptions to determine the class to which an experiment belongs. The `purify-by-class` method needs two input models: `new experiment` and `class descriptions`. The `New experiment` model contains the description of a sample from which a protein has to be purified. The `class descriptions` model contains the descriptions of the classes to which a purification experiment can belong. This model is not provided by the domain expert, so during the KM analysis a KA-Task has to be associated to it (see figure 8.2). This KA-Task is solved using a learning method, called `induce-classes`, that induces the descriptions of the classes from the experiments contained in the `experiments` model.

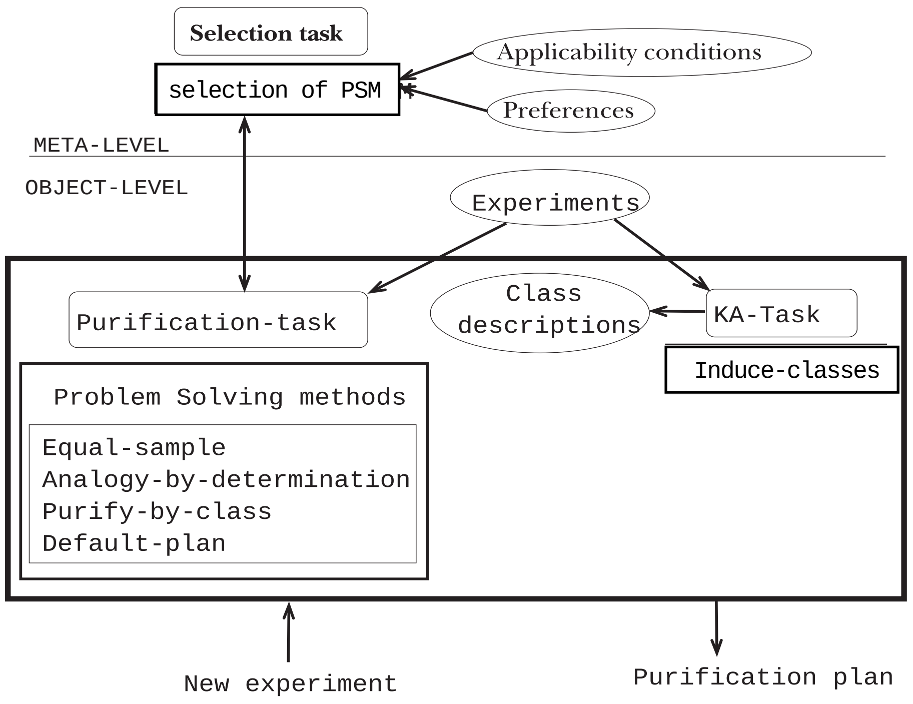


Figure 8.2. The CHROMA architecture.

Finally, strategy M4 has been modelled using the `default-plan` method: domain-specific method acquired during the knowledge engineering stage. This method is based on statistical analysis of purification experiments and provides a starting point for the trial and error process of the M4 strategy.

Next step is to decide which of the four methods above is the most appropriate to achieve the `purification` task. According to the framework described in chapter 3, the selection of the appropriate method achieving a task can be delayed until the Problem Solving phase. In CHROMA, this selection is made using a task at the meta-level (figure 8.2). Thus, when a new experiment has to be solved, it is analysed at the meta-level in order to choose which of the four methods above is the most appropriate.

Input models of the `purification` task are the `experiments` model (that contains purification experiments already solved), and the `new experiment` model (that contains the description of experiment to be purified). During the KM analysis, four PSM have been associated to the `purification` task. The selection of one of them is delayed until the Problem Solving phase.

Given an experiment to be solved, the `selection` task at the meta-level of `purification` task analyses the `applicability conditions` model and the `preferences` model (see section 2.1.1 in chapter 3) and selects the most appropriate PSM. In particular, when the selected method is the `purify-by-class` method, the `class descriptions` model is necessary. During the KM phase we have defined a KA-Task that uses a learning method, called `induce-classes`, to acquire this model.

In the next sections each method is explained in detail.

2.2.1. The `Equal-Sample` Method

The `equal-sample` method is a case-based method composed by two tasks : `retrieve` and `reflect` (figure 8.3). Method for `retrieve` task is `retrieve-by-pattern` that is a NOOS built-in method that retrieves from the memory those cases that are subsumed by a pattern. In this situation, the `equal-sample` method takes the `description` of the current experiment as a pattern and searches the base of cases for experiments having at least that description. In other words, the `retrieve` task retrieves, if exists, a past experiment whose `description` is subsumed by the current experiment description. The `reflect` task completes the construction of the `Solved Episode` model of the new experiment. This task assigns as value of the `solution` feature of the new experiment, the purification plan of the experiment retrieved by the `retrieve` task.

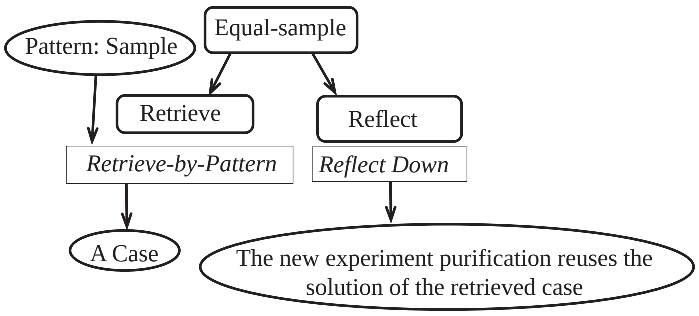


Figure. 8.3. Task-method decomposition of `equal-sample` method.

The `equal-sample` method fails if there is no experiment having the same description than the new one. The `equal-sample` method is useful to solve routine purifications with commonly occurring samples and proteins and assures a successful solution (since it deals with "identical" problems).

2.2.2. The `Analogy-by-Determination` Method

The `analogy-by-determination` method is a case-based method for solving the `purification` task. This method uses the domain knowledge embodied in a *determination* stating that the correct plan for the `purification` task is determined by the protein to be purified.

Determinations are functional dependencies that can be used to justify analogical reasoning (Russell, 1990)[3]. In the `analogy-by-determination` method the determination used states that the solution for the `purification` task depends on the value of the `protein` feature. The method can justify a purification plan as solution from the fact that a precedent case with the same protein has successfully used that plan.

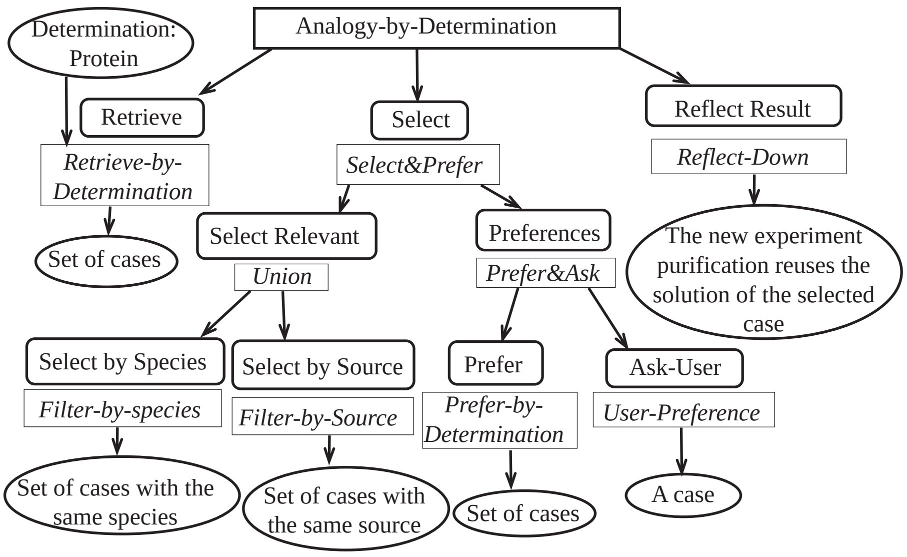


Figure. 8.4. Decomposition of `Analogy-by-determination` method.

The `analogy-by-determination` method is composed by three tasks: `retrieve`, `select` and `reflect` (figure 8.4). The `retrieve` task uses the method `retrieve-by-determination`, that searches in the case base for those experiments purifying the same protein as the current problem. When it retrieves more than one experiment, the `select` task uses the method `select-&-prefer` in order to select only one experiment. The `select-&-prefer` method has two subtasks: `select-relevant` and `preferences`. From the set of experiments obtained by the `retrieve` task, the `select-relevant` task selects the subset of experiments that have either the same `species` or the same `source` as the new experiment.

If the `select-relevant` task retrieves more than one experiment, the `preferences` task is used to select only one of them. The `preferences`

[3] The classical example of an analogy justified by a determination is the following: the usual language spoken by a person is determined by the person's nationality. We know a case, Janos, that is Hungarian and speaks Magyar. The language of another person can be solved by an analogy-by-determination method. If this method is used for the task of determining the language of a person that happens to be Hungarian then it concludes that he speaks Magyar because of the determination and the Janos precedent.

task uses the `prefer&ask` method that decomposes it in two subtasks: `prefer` and `ask-user`. The `prefer` task has as criterion that of preferring those experiments using a species of the same kingdom as the new experiment. If the `prefer` task obtains more than one preferred experiment, the `ask-user` task presents them to the user who can choose one of them.

Finally, the `reflect` task completes the `New solved episode` model by instantiating for the `solution` feature of the current experiment the purification plan of the precedent selected by the previous task.

The `analogy-by-determination` method fails when there is no experiment in the case base purifying the same protein as that of the current experiment (i.e., it fails when there is no experiment in the case base that satisfies the protein determination).

2.2.3. The `Purify-by-Class` Method

The `purify-by-class` method uses as input models the `new experiment` model and the `class descriptions` model. During the KM analysis of the domain, we have identified sets of experiments following the same purification plan. We have considered that those experiments using the same purification plan constitute a class. The description of a class is determined by the common characteristics of the experiments using the same purification plan. We can define a KA-Task to acquire the class descriptions during the problem solving phase. We associate to this KA-Task a learning method, called `induce-classes`, that induces the descriptions of the classes from the case base (this method is explained in the next section).

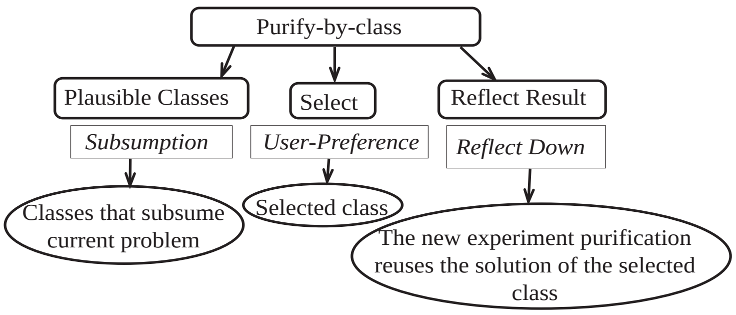

Purify-by-class
Plausible Classes
Select
Reflect Result
Subsumption
User-Preference
Reflect Down
Classes that subsume current problem
Selected class
The new experiment purification reuses the solution of the selected class

Figure. 8.5. Task-method decomposition of `purify-by-prototype` method.

The `purify-by-class` method is composed of three tasks: `plausible-classes`, `select` and `reflect` (figure 8.5). The `plausible-classes` task selects, using the `subsumption-matching` method, only those classes whose description subsumes the new experiment. When more than one solution class is obtained by `plausible-classes`, the `select` task, using the `user-preference` method, asks the user to select one of them. As in the `analogy-by-determination` method, the final decision has to be taken by the user and NOOS supports the browsing and inspection of the alternatives in a graphical interface. So the user can examine in detail the selected classes (and also the precedents in the classes) before choosing one class.

The `reflect` task completes the `Solved Episode` model of the current experiment taking as value for the `solution` feature the purification plan associated to the selected class.

The `Purify-by-class` method fails when there is no description class subsuming the current experiment.

2.2.4. The `Induce-Classes` Method

This method is associated to the KA-Task to acquire the `class descriptions` model. A class C_i contains experiments having the same purification plan. From the samples of the experiments belonging to C_i, the `induce-classes` method induces a description $D_i = \{p^j{}_i\}$ (that is a disjunction of descriptions) for C_i. The description of a class C_i is obtained by generalising the samples of the examples belonging to C_i.

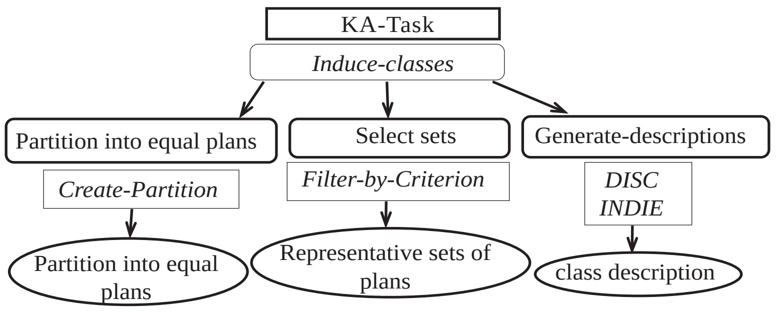


Figure. 8.6 Task-method decomposition of `induce-classes` method.

Thus, a new experiment E belongs to a solution class C_i if there is a description $p^j{}_i$ in D_i subsuming E. Therefore, E may be purified using the purification plan associated to C_i.

The `induce-classes` method is composed by three subtasks: `partition-into-equal-plans`, `select-sets` and `generate-descriptions` (figure 8.6). The method of `partition-into-equal-plans` task divides the case base in sets containing experiments with the same purification plan. Because some of the formed sets may have few elements and it is not desirable to make induction from them, the `select-sets` task obtains only those sets having a number of elements above a threshold. The goal of the `generate-descriptions` task is to generate a description for each solution class selected by the `select-sets` task. To solve the `generate-descriptions` task any inductive method explained in Part II can be used. Thus, the model of each solution class is a disjunction of descriptions. Each description is a feature term having the same features as an experiment, i.e. *description-experiment* and *purification*, but the values of them are generalised. For instance, the description in figure 8.7 can be applied if the new experiment has a *protein* belonging to the Dehydrogenases family, the sample *species* is an animal, and the sample *source* is liver. In other words, according to the experiments in the case base, any Dehydrogenase protein may be purified in any tissue of an animal species using a purification plan composed of four steps: Precipitation, Gel Filtration, Ion Exchange and Affinity.

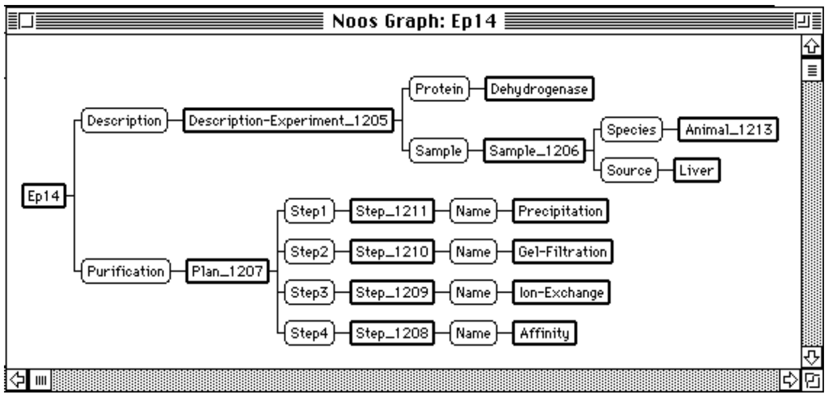


Figure 8.7. The *experiment-pattern* EP14 is the description for the solution class containing experiments that use a purification plan composed by four steps: a Precipitation, a Gel Filtration, an Ion-Exchange and an Affinity.

2.2.5. The `Default-Plan` Method

The `default-plan` method is a domain-based method that can be used indepently of the *protein* and the *sample* of an experiment. This default purification plan has been obtained from a statistical study (Janson and Ryden) where many purification plans were analysed. The conclusion of this study was that the most frequently used chromatographic techniques in each step of a purification plan are the following: 1) Clarification, 2) Ion-Exchange, and 3) Gel-Filtration. This purification plan is useful when there is no literature on a particular protein subject. The use of this purification plan implies a costly adjustment in the laboratory.

This method is equivalent to design a purification plan according to the characteristics of both each chromatographic technique and the protein to purify. The advantage is that the `default-plan` method focuses the user on adjusting the parameters of the suggested techniques. In that way the time to design an appropriate purification plan is considerably reduced.

2.3. Integration

During the KM analysis of the domain, four PSM have been associated to the `purification` task. CHROMA uses a lazy problem-centred selection to decide which method to use to solve the `purification` task for a new experiment. The `selection` task at the meta-level of `purification` task performs the selection of the appropriate method. Before the selection of some method, the new experiment is analysed. From this analysis and the models `Applicability Conditions` and `Preferences`, the `selection` method associated to the `selection` task chooses an appropriate method. If the selected method does not achieve a solution, another method has to be tried.

Let us suppose that in the CHROMA application the methods `equal-sample`, `purify-by-class`, `analogy-by-determination`, and `default-plan` are sequentially tried in this order. If a new experiment wants to purify a protein that is not used in any experiment of the base of cases, the only applicable method is `default-plan`. Using the sequential order, all the methods have to be executed (and fail) before to obtain the solution from `default-plan`.

The KM analysis of the domain suggests a more intelligent strategy to select the appropriate method. We propose to use a lazy problem-centred selection of the method taking into account the `Applicability Conditions` model and the `Preferences` model (see section 2.1.1 in chapter 3). In particular, the `Applicability Conditions` model in CHROMA contains the following knowledge:

- If the protein of the current problem is not purified in any experiment in the case base (`Problem Solved Episodes` model), the only applicable method is `default-plan`.
- If there is no experiment in the case base using the same sample that the current problem the applicable methods are the `purify-by-class` method, the `analogy-by-determination` method and the `default-plan` method.
- If the sample of the current problem does not satisfy any class description, the applicable methods are the `analogy-by-determination` method and the `default-plan` method. As we will see later, to evaluate this condition CHROMA needs an additional model called `control sample`.

The `Preferences` model contains preferences provided by the domain expert in order to choose one method if more than one is applicable. In CHROMA the `Preferences` model contains the following preferences:

1) If applicable, `equal-sample` is preferable to others (since identical precedent assures an appropriate solution)

2) `default-plan` is the less preferable

3) `analogy-by-determination` and `purify-by-class` are equally preferable if both are applicable.

The lazy problem-centred strategy has been implemented using a `selection` task at the meta-level of the `purification` task. The `selection` task has as input the `control sample` model that contains the description of a sample S. Each feature A of the sample S has as values the disjunction of the values that A takes in all the case base experiments. The `selection` task is solved using the method described in figure 8.8. Thus, if there is no experiment in the case base using the current protein, the purification plan is always to be obtained using the `default-plan` method. If the protein was already used and there is an experiment having the same sample that the new one, the `equal-sample` method can be used. If the new experiment belongs to some solution class, the `purify-by-class` method can be used (also the `analogy-by-determination` method could be used). Otherwise, the new experiment only can be solved using the `analogy-by-determination` method.

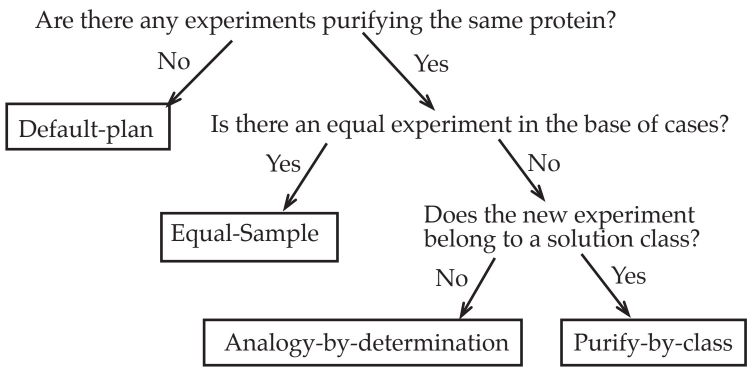

Are there any experiments purifying the same protein?
No
Yes
Default-plan
Is there an equal experiment in the base of cases?
Yes
No
Equal-Sample
Does the new experiment belong to a solution class?
No
Yes
Analogy-by-determination
Purify-by-class

Figure. 8.8. Decisions taken by the method used by `selection` task at the meta-level of `Purification` task.

Notice that using only two methods (`analogy-by-determination` and `default-plan`), CHROMA can always solve the `purification` task. We can thus ask, is it really better to use four methods when two are enough? The use of several methods requires the definition of some criterion to select the most appropriate. The most simple criterion is the sequential one, i.e. applying each method in a predetermined order. A method is only applicable if the previous one has failed. CHROMA has a more intelligent strategy using a `selection` task at the meta-level. When using the `selection` task at the meta-level is better than using the sequential strategy? In the next section several configurations of CHROMA are evaluated in order to answer the questions above.

3. Evaluation

In this section, four configurations of CHROMA are compared. A *configuration* is the set of both the problem solving and the learning methods that can be used. The `default-plan` method is included in all the configurations in order to assure that CHROMA always solves the `purification` task. Only one configuration may have a `selection` task at the meta-level selecting the appropriate method in a lazy problem-centred way. In the following the four CHROMA configurations are described.

Configuration 1. There are two problem solving methods: `analogy-by-determination` and `default-plan`. Using this configuration, new experiments using proteins already existing in some experiment already performed will be solved using the `analogy-by-determination` method. Other experiments will be solved using the `default-plan` method.

Configuration 2. This configuration is composed by the problem solving methods `purify-by-class` and `default-plan` and by an inductive learning method (any of the explained in Part II) associated to `purify-by-class` method. In particular, we have evaluated this configuration using the INDIE method with the generalisation post-process. Using this configuration, only new experiments subsumed by some solution class description will be solved using the `purify-by-class` method. Other new experiments will be solved using `default-plan` method.

Configuration 3. The `purification` task is solved by sequentially selecting the methods `purify-by-class`, `analogy-by-determination` and `default-plan`. In this configuration, a method is selected only when the previous one has failed to find a solution. The rationale behind this ordering is to use first the more specific methods. This sequential strategy is the most used in systems having more than one problem solving method.

Configuration 4. This configuration uses the same methods that configuration 3 but also uses the `selection` task at the meta-level of the `purification` task.

The empirical comparison of the configurations is made taking into account the mean time used for solving 25 new experiments. The solution quality is not analysed because it is difficult to evaluate without actually performing the purification experiment in the laboratory. However, we know that the purification obtained from the `default-plan` method is the less desirable because it is too general and adjusting its solution requires many laboratory time. Therefore, those configurations that minimise the use of the `default-plan` method are preferred.

The `equal-sample` method does not appear in the configurations because the `selection` task cannot decide if it is applicable without executing it. If the `equal-sample` method is taken into account, a constant (the mean execution time of the `equal-sample` method) has to be added to the total time obtained for each configuration.

The evaluation of each configuration is made using a training set containing 108 solved experiments from which 14 solution classes with their respective descriptions can be induced. Testing involves 25 new experiments that are presented to CHROMA in 15 test sets with different random orders; the obtained values are the mean values averaged over these 15 test sets. From these 15 test experiments, 7 can be solved using `purify-by-class`, 12 by `analogy-by-determination` and 6 by `default-plan`. The results of the evaluation are the following.

Configuration 1. Six of the 25 new experiments can be solved using the `default-plan` method and the remaining ones are solvable using the `analogy-by-determination` method. Mean time of this configuration in solving one experiment is 1.338 seconds.

Configuration 2. From the 25 new experiments that are proposed to CHROMA, only 7 can be solved using `purify-by-class`. This means that other 18 new experiments are solved using the `default-plan` method that is not enough accurate and it is less desirable than configuration 1. The mean time of configuration 2 is 1.221 seconds.

Configuration 3. In this configuration the methods are sequentially tried until one that solves the new experiment is found. Thus, the time for solving an experiment is increased by the time expended executing methods that eventually fail. For instance, from the 25 test experiments, 6 can only be solved by the last method, `default-plan`; in these 6 experiments, the mean time is the sum of the execution times for the three methods. From the remaining 19 experiments, only 7 are solved by the `purify-by-class` method, and the remaining 12 are solved by the `analogy-by-determination` method. The mean time of configuration 3 is 1.352 seconds.

Configuration 4. The main idea of configuration 4 is to choose in a lazy problem-centred way the appropriate method to solve a new experiment. The advantage in respect to configuration 3 is that only one method will be completely executed. Using the `selection` task, time requirements are more stable because newly stored cases only increase the problem solving time during the retrieval and the selection subtasks of the `analogy-by-determination` method. Mean time of this configuration is 1.022 seconds.

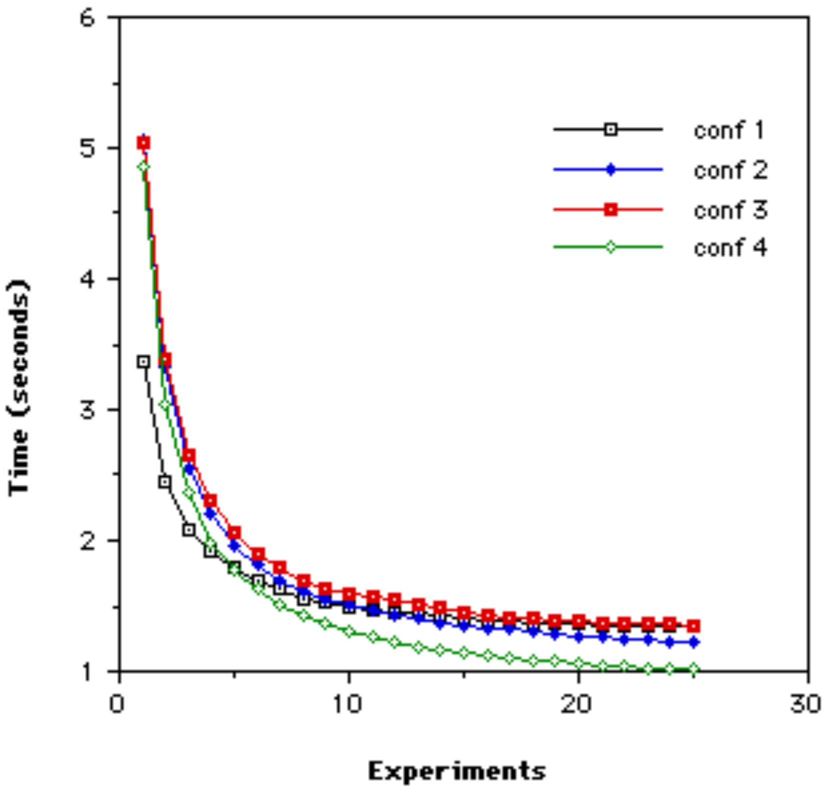


Figure 8.9. The evolution of problem solving time in four different configurations of CHROMA. The values shown are averaged over 15 trials.

Figure 8.9 shows the time evolution of the configurations above. The time is initially high due to NOOS, since internal structures are created in a lazy way. This time decreases quickly and later stabilises. In the stable stage, the best configuration is clearly Configuration 4. Also, we can see that Configuration 3 (using three PSM and a sequential selection them) does not improve Configuration 1 and Configuration 2. Configuration 1 is worst because new solved experiments are added to the case base and thus the retrieval time increases. This increment only appears in Configuration 4 when the `analogy-by-determination` method is chosen to solve a new problem, other PSM are not affected.

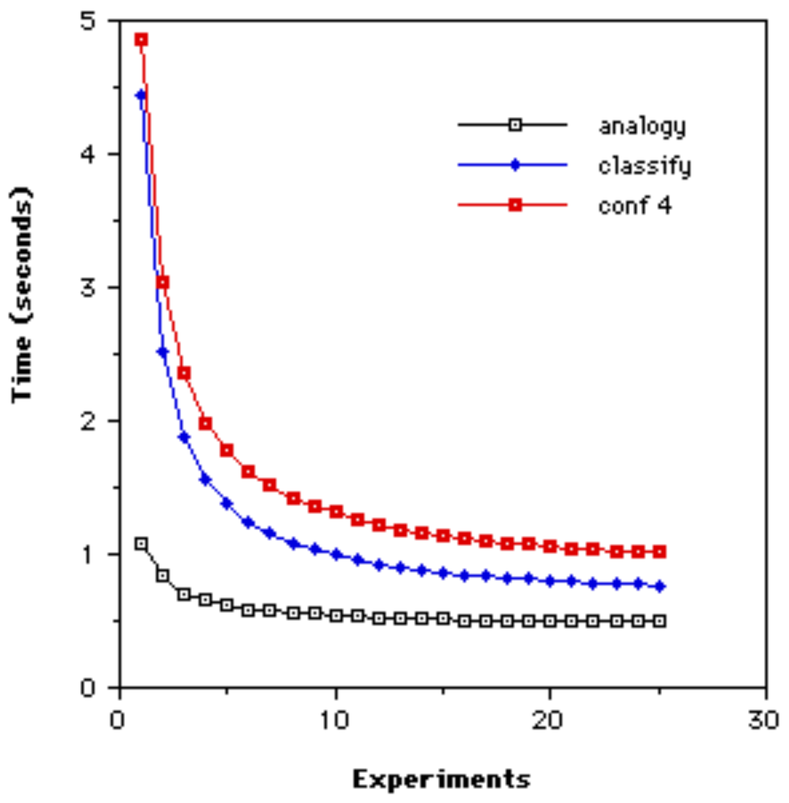


Figure. 8.10. Comparison of the time evolution of configuration 4 with configurations having only one method (`analogy-by-determination` or `purify-by-class`).

Other evaluations have been made comparing Configuration 4 with having only one method (`analogy-by-determination` or `purify-by-class`). Figure 8.10 shows the result of this comparison. The `Analogy-by-determination` method takes a mean time of 0,497 seconds, whereas the `purify-by-class` method takes 0,764 seconds. Clearly, having only one method is more efficient but when the method fails CHROMA cannot provide a solution. Thus, Configuration 4 takes more time but always provides a plan to purify a new experiment. For instance, to use only the `purify-by-class` method implies that from the 25 new experiments of our test, CHROMA produces no solution for 18 of them.

There are several conclusions from this empirical analysis. The first one is that the `selection` task at the meta-level not always deteriorates the CHROMA performance. In other words, the use of the `selection` task tends to increase the predictivity without losing efficiency. A second conclusion is that the mean time of Configuration 4 is clearly better that the mean time of Configuration 3, that is one of the most common strategies. The mean time of Configuration 1, initially best, tends to worsen when the number of cases in the base increases. Instead the mean time of Configuration 4 appears as more stable when more problems are solved.

4. Conclusions

The CHROMA application has been developed using the framework proposed in chapter 3. During the KM analysis we have modelled the main task, `purification`, as a task that can be achieved by four methods. The analysis of these methods has identified the models required to achieve the `purification` task. The acquisition of one of these models, `description classes`, is delayed to the Problem Solving phase. To this purpose, we have modelled a KA-Task that using an inductive learning method induces this model from the experiments in the case-base.

During the KM analysis we have also determined under which conditions each method associated to the `purification` task can be applied and which are more preferable. Therefore, this analysis has shown a strategy to select the appropriate method that is more intelligent that the sequential strategy. In particular, we have defined a `selection` task at the meta-level of `purification` task. This task is solved by a meta-level method called `selection`. The `selection` method uses the knowledge acquired during the KM about the applicability conditions of the methods and the preferences among them, and also analyses the experiments to be solved in order to select a method for the `purification` task. The KM analysis of the domain is a key issue in determining which selection strategy may be more efficient.

A future line of research is to investigate how to automatically learn the selection strategy based on features of previously solved problems and on features of the available methods.

There is some work dealing specifically with the combination of case-based methods and knowledge produced by inductive learning methods. The KATE system (Manago, 1989) induces a decision tree and combines decision-tree classification with case-based classification. The combination of methods is fixed: decision-tree classification is tried first and if it "fails" (for instance, because of a missing attribute) then a case-based method is used. The INRECA project (Manago et al., 1993) follows a similar approach integrating the induction of decision trees (using KATE) and case-based reasoning using PATDEX. Currently KATE and PATDEX are able to interchange results through the format of the CASUEL language, but the combination of both methods is fixed. The integration of the multiple inference methods is easily realised by the support given by the NOOS language to the knowledge modelling analysis of the CHROMA application. As we have seen all methods (case-based, inductive, knowledge-based) can be described in our knowledge modelling framework and then implemented using the NOOS language. This is a tighter integration than other proposals for integrating inductive and case-based methods; for instance the INRECA project is based on the establishment of the syntax of an interchange format called CASUEL. Different modules (CBR, induction) read from and write to this format but each module uses a different representation language.

Concerning the chromatography application domain, there is only one system called FPLCassistant™ from Pharmacia (Osterlund, 1993). This application uses as domain knowledge textbook information about the conditions under which each chromatographic technique is useful. FPLCassistant™ advises the user each step of the purification plan. The user has to determine parameters of the experiment such as volume of the sample, pressure, pH, pI, etc. Nevertheless, the determination of some of these parameters is not an easy task. In particular, the determination of the pI (isoelectric point) requires a great laboratory effort, sometimes bigger than adjusting a purification plan already used in another experiment. In fact, this is the methodology followed by an expert, and also by CHROMA, i.e. the purification plan used in an experiment with a similar sample is adjusted in order to purify the current sample.

Chapter 9

SPIN: A Tool for Marine Sponges Identification

1. Introduction to Marine Sponges Domain

The identification of specimens is a very common task in biological research. There are several types of biological studies, for instance ecological investigations, that need a taxonomic analysis of organisms. Frequently, an error in the identification of the organisms invalidates the whole study.

Systematics is the part of the Biology having as goal the identification of organisms. Typically, researchers in Systematics support researchers in other fields of Biology in characterising and identifying the organisms they deal with. So, a tool supporting the specimen identification to non-experts in Systematics may be interesting since each expert could do their own research.

Marine sponges (phylum Porifera) are relatively little studied and most of the existing species are not yet fully described. The identification of marine sponges is specially complex and often the support of an expert is necessary. Moreover, sponges are genetically much more diverse than other marine invertebrates. This high variability is also present within species due to their capability of adaptation to environmental conditions. As example of the complexity in the identification of marine sponges, we want to remark than some researchers have assumed the discovery of a new specie whereas it really was a morph of an already known species.

Historically, the identification of marine sponges has been based on characteristics of the skeleton. Nevertheless there are other biological and cytological characteristics that have not been sufficiently studied. Main problems are due to the morphological plasticity of its species, to the incomplete knowledge of many of their biological and cytological features, to the frequent description of new taxa, to the variety of specific terms applied to homologous characters and, conversely, to the occurrence of single terms that name analogous characters. Since taxa are closely alike, controversy around the species delimitation or even around higher taxonomic levels of the classification is common. More details about the morphology, behaviour and systematic of the sponges can be found in appendix C.

But identification needs classification, and in the Porifera phylum it is not clear how the different taxa are to be characterised. At present, the general requirements for accurate identification are concerned with descriptional characters. In taxonomy, characters are used to discriminate between taxa. A character should be discontinous enough to be diagnostic for a taxon[1]. Indeed, characters should be stable, that is, genetically spread but independent of external influence. Or, if influenced by the environment, their incidence should be statistically established before being used as diagnostic characters. These are desired requirements but in many taxonomic domains they are not accomplished. In other words, *Taxonomic knowledge* is not simply a hierarchy of taxa but it is concerned with a highly complex theory that emerges from the study of the characters, their variability, their phylogenetic significance, etc. In many taxonomic groups, there is not enough knowledge to build a definitive theory.

The basic goal of taxonomic studies of living organisms is to establish their phylogenetic relationships on the basis of observable features and to build up a store of information from which deductions can be made about biological processes. At present, this goal is difficult to achieve in many taxa of the Porifera due to the lack of well-established taxonomic theories, the subjective interpretation of features and the inherent imprecision of natural language, which has been traditionally used to express the taxonomic knowledge.

The taxonomic classification of any group of organisms has the aim of establishing a hierarchical ordination of taxa with two major goals: (1) to provide a framework for making biological statements and generalisations and (2) to serve as an information storage system. There are three different approaches followed by taxonomists when trying to build a classification. The most objective method for achieving this classification is called *phenetical* or *numerical taxonomy*. This method considers all the characters that are available without weighting them. This involves numerical procedures on distance or correlation matrices among the organisms to be classified. The classic methodology for approaching this classification is called *cladistics* or *phylogenetic systematics* and searches for monophyletic classes. Lastly, a third approach is made by

[1] Characters are said to be *diagnostic* for a taxon when their presence in a particular state confirms the identification of members of that particular taxon.

the *evolutionary systematics* school. They interpret the observed similarity in the light of its evolutionary history. As in the case of the previously mentioned school, it aims to build classes on the basis of a common ancestor but permits groups that are not entirely monophyletic.

Expert research usually focus on one or several portions of the Porifera taxonomy. Thus, expertise is scarce and phylum knowledge is usually scattered among several experts. In fact, taxonomic knowledge in any group depends on the past work of a systematic biologist in this group and the expertise knowledge of the present experts, taking into account current scientific trends, is to be lost when they retire. Recently, several sponge systematists are using database systems to store either bibliographic references, species names or biogeographic information. To date, these databases have been individual efforts of creating a tool useful for taxonomic research, mainly concerned with the access to specialised literature, but not with any identification purpose.

There are two general methods used in the identification of specimens: elimination or matching. *Elimination* consists in successively observing a new character in the unknown specimen and eliminating the taxa that do not present such value for that character. *Matching* involves a direct comparison of the specimen with the standard description of the taxa. This is usually based on a similarity criterion related to the number of similar (or dissimilar) characters. Obviously, analogy of a number of characters does not necessarily mean a common taxon assignment unless these characters are diagnostic for that taxon. Both of these methods assume that the possible taxa solutions are known. The experts' identification skills seem to combine elimination and matching to result in a particular identification procedure that is based mainly on the expert's heuristics.

The *diagnostic* or *dichotomous keys* (an elimination method) and the *multiple-entry key* are methods traditionally used to identify sponges. The main shortcoming that they present is that if a specimen is partially described and a key is not present, the specimen cannot be identified. There are also on-line programs that emulate the identification methods and improve them in the selection of more accurate characteristics to perform an identification. Nevertheless, most of the on-line programs have difficulties to deal with uncertain and unknown data, even though some of them incorporate different facilities to solve special situations in the identification process.

Expert System technology also provides a means for building computer assisted identification tools. Some expert systems dealing with taxonomic knowledge have been developed. The more illustrative are to be briefly mentioned. Wooley and Stone (1987), in a general discussion of the advantages of expert systems for taxonomic identification, report on the development of an expert system to identify 12 species of the genus *Signiphora* Ashmead (Hymenoptera:Signophoidae). A case-based expert system dealing with the genus *Hyalonema* Ijima (Porifera:Hexactinellida) is briefly reported in Conruyt et al., (1993). It was build from about a hundred case descriptions and its identification scope is the level of subgenus (12 taxa). In this work the emphasis is put on the modelling of an expert's knowledge rather than on the usefulness of the identification tool. An interesting expert system application dealing with Mediterranean zooplankton (about 100 species) is presented in Thonnat and Gandelin (1988). This program consists of an automatic system that searches for complex shapes representing the main groups of zooplankton. Each group is represented in two dimensional images and a classification expert system that identifies the species. In this case, both reasoning and pattern matching techniques are used. SPONGIA (Domingo, 1995) is a knowledge based system developed using the Milord II programming environment (Puyol, 1994). SPONGIA deals with the identification of sponges from the Atlanto-Mediterranean biogeographical province. It covers the identification of more than 100 taxa of the phylum Porifera from class to species. The use of fuzzy logic makes possible an accurate representation of the imprecise knowledge which constitutes the classificatory theory of Porifera to a large extent. SPONGIA also provides the user with some means of expressing his state of knowledge with accuracy.

In this chapter we describe SPIN, a tool capable to identify marine sponges. This identification can be made using domain-specific and case-based methods. A main contribution of SPIN is the capability to deal with partially described specimens.

2. Description of the SPIN Application

In this section we use the framework explained in chapter 3 to develop SPIN, a KS capable to identify the taxa to which a marine sponge belongs. Taxa are organised in five taxonomic levels: *class, order, family, genus* and *species*. A specimen is completely identified if it has been included in one taxon of each taxonomic level.

During the KM analysis we have determined that there are to ways in which a specimen E can be identified. The first one is to compare the characteristic description of each taxon to the description of E. As a consequence, E can be said to belong to a taxon T when E shares the characteristic features with the description of T. The second way is to search in the literature for descriptions of already identified specimens, until a specimen, say P, very similar to E is found. In such situation E can be classified as belonging, at each taxonomic level, to the same taxa that P.

In this section we describe the models used in the SPIN application. Then we show how the `identification` task is solved. The SPIN implementation is made using the NOOS language described in chapter 3.

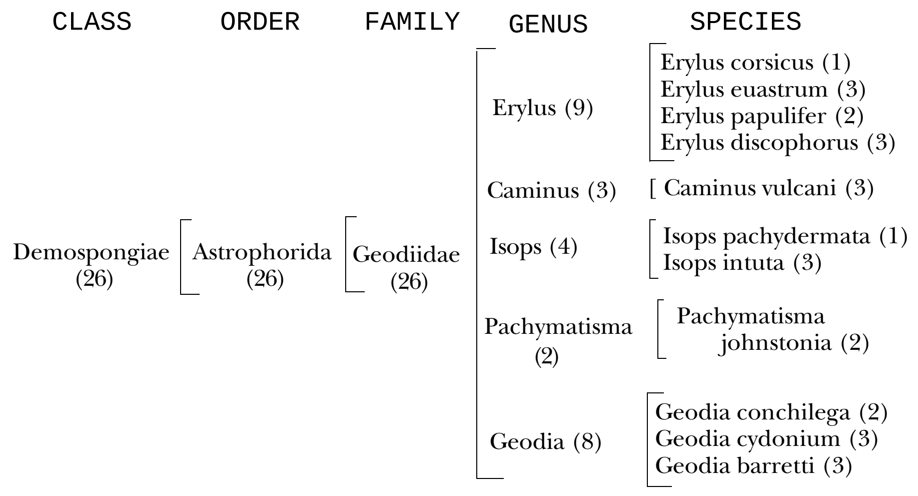


Figure 9.1. Part of the Porifera taxonomy to which belong the 26 sponge specimens used by SPIN. The number means how many specimens has each taxon.

2.1. Models in SPIN

The identification of a new sponge is achieved in SPIN by means of the `identification` task (explained in the next section). The `identification` task needs two input models: one that contains the description of a sponge to be identified (`new sponge` model) and the other containing sponges already identified (`classified sponges` model).

In particular, the `classified sponges` model contains the description of 26 sponge specimens. These descriptions are a subset of the specimens that have been used to test the SPONGIA system (Domingo, 1995). We have not used all the features reported in the literature for each sponge specimen but only those features used by SPONGIA. In other words, let us suppose that a sponge E in the literature has been described by means of a set of features F. During the execution of SPONGIA only a subset F' of these features have been used to identify E. Thus, E is described in SPIN using only the subset F'. This is an important remark since the SPIN behaviour could have been different if each specimen were described using all the known features F. On the other hand, results of both SPIN and SPONGIA may be compared.

A specimen can be identified as belonging to taxa at five different taxonomic levels (see figure 9.1): *class, order, family, genus* and *species.* The 26 available specimens belong to the *Demospongiae* class, *Astrophorida* order and *Geodiidae* family, thus SPIN can only identify sponges at the level of *genus* and *species.* Nevertheless, SPIN could identify sponges belonging to levels higher than *genus* if the `classified sponges` model were enlarged.

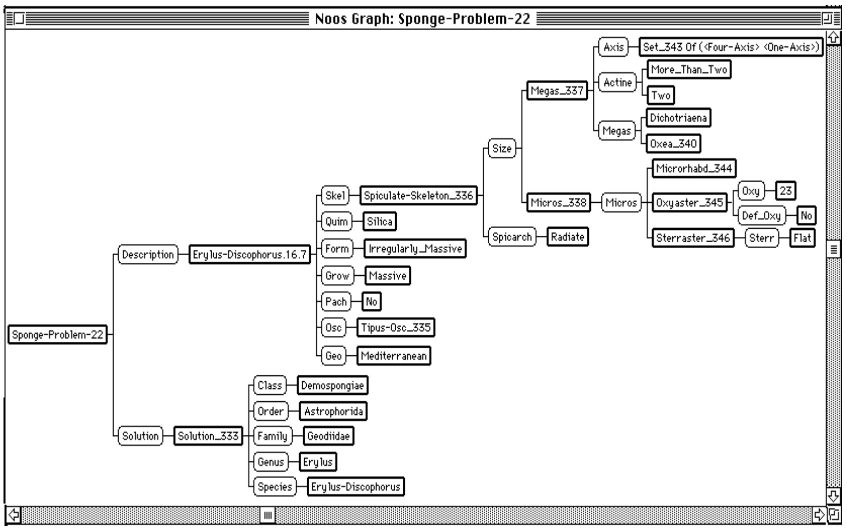
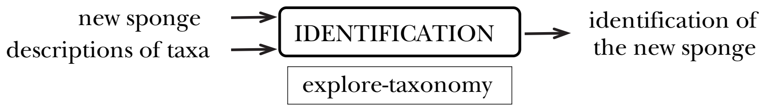


Figure 9.2. Description of a *sponge-problem* in SPIN. A *sponge-problem* is composed of a description of a sponge specimen and its classification in five taxonomic levels: *class, order, family, genus* and *species.*

Each sponge in the `classified sponges` model is a feature term belonging to the sort *sponge-problem.* Objects of this sort have two features (figure 9.2): `description` and `solution`. `Description` is a feature having as value an object belonging to the sort *sponge.* Objects belonging to the *sponge* sort have a different number of features according to known characteristics of each specimen (a sponge may be partially described). The `solution` feature of an *sponge-problem* contains the identification of the described sponge in the taxa *class*, *order*, *family*, *genus* and *species.* The sponge to be identified (contained in the `new sponge` model) only has the `description` feature. The `identification` task assigns a value to the `solution` feature of the new sponge.

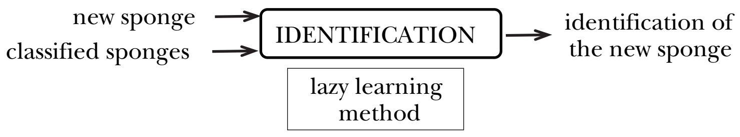
2.2. Solving the `Identification` Task in SPIN

Commonly, the identification of specimens is made from the descriptions of the taxa. Therefore, given a sponge specimen E, a strategy for achieving the `identification` task is to explore the taxonomy and to find, for each taxonomy level, one taxon in which E can be included. This goal is achieved by a task called `identification` that, in addition to the model containing the description of E, also needs another model containing the descriptions of the taxa. We can thus model the `identification` task in the following way:



i.e. the `identification` task may be solved using the `explore-taxonomy` method that searches in each taxonomy level for those taxa whose description subsumes the description of the sponge to be identified.

However, as we have explained in previous section, there is no agreement among the experts about which are the characteristic features of the taxa. Nevertheless, there are many sponges whose identification is not discussed, so they could be used to identify new sponges. In other words, a new sponge can be identified using a lazy learning method that uses as input a model containing a set of sponges and their classification at each taxonomy level. In such situation, the modelling of the `identification` task can be the following:



Summarising, given the description of a new sponge E, the goal of the `identification` task is to identify the *class, order, family, genus* and *species* to which E belongs. There are two methods that can be followed to identify a new sponge:

1. A lazy learning method searching in the case base for the most similar sponge and identify the new one as belonging to the same taxa that the precedent. We have used `LID` to implement this CBR method

2. to use the general description of each taxa to find one taxon of each level in which the new sponge can be included. We call this method `explore-taxonomy` and it is explained presently.

Notice that the `descriptions of taxa` model, necessary to use the `explore-taxonomy` method, are not directly available from experts, as we have already explained. An alternative is to learn the descriptions of the taxa model from the description of the already classified sponges. So, during the KM analysis we have defined a KA-Task for `descriptions of taxa` model, and an associated inductive learning method capable to acquire the descriptions of taxa.

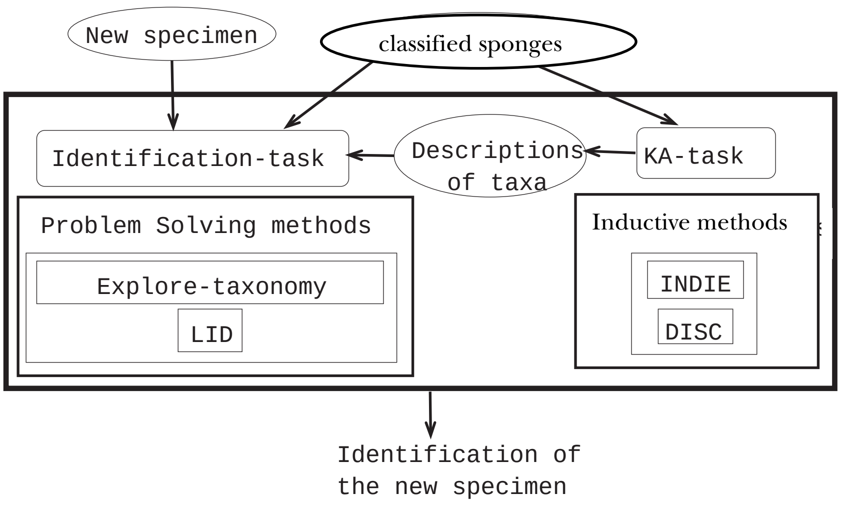


Figure 9.3. Structure of SPIN. The main task, called `identification` task, can be achieved using two methods: `explore-taxonomy` or `LID`. The `KA-Task` can be achieved using the INDIE or DISC methods. Notice that learning takes place not only for the KA-Task, but also inside the lazy learning process of LID.

Figure 9.3 shows the structure of the SPIN application according to the knowledge modelling analysis previously explained. SPIN has not a selection method at the meta-level selecting the appropriate method to identify a new specimen. Instead, SPIN uses a sequential strategy to solve the `identification` task. This strategy consists of to use first the `explore-taxonomy` method and, if it fails, a lazy learning method such as LID will be used. The domain expert has provided only descriptions of sponge specimens and their classification (the `classified sponges` model). Therefore, when the `explore-taxonomy` method is used, there are not available the descriptions of the taxa (the `descriptions of taxa` model). The acquisition of the `descriptions of taxa` model is made using a KA-Task that has associate two inductive learning methods. The inductive learning method to be used to solve the KA-Task is selected by the user.

In the next sections we explain the `explore-taxonomy` method and how SPIN integrates learning and problem solving. Since LID has already been explained in detail in chapter 7, we will not repeat this explanation here.

2.2.1. The `explore-taxonomy` Method

The `explore-taxonomy` method may be modelled as having five subtasks (one for each taxonomy level): `identify-class`, `identify-order`, `identify-family`, `identify-genus` and `identify-species`. Each one of these tasks is solved by the same method, called `identify-by-subsumption`. Given a taxonomy level TL (i.e. *class, order, family, genus* or *species*), the `identify-by-subsumption`. method uses the descriptions of the taxa in TL to identify the new specimen.

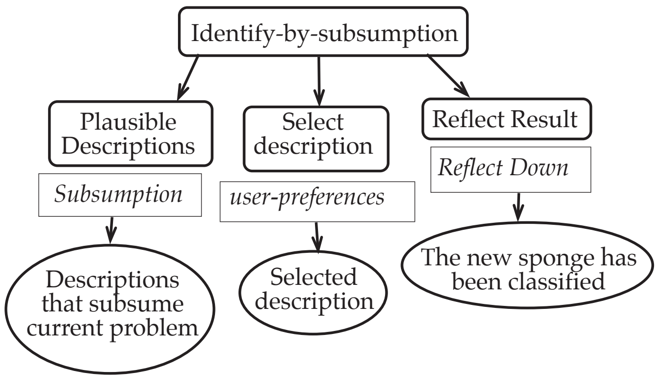


Figure. 9.4. Task-method decomposition of `Identify-by-subsumption` method.

The `Identify-by-subsumption` method is composed by three tasks (figure 9.4): `plausible-descriptions`, `select-description` and `reflect-result`. The `plausible-descriptions` task selects, using the `subsumption` method, only those taxa whose description subsumes the description of the new sponge. The subsumption method uses the `descriptions of taxa` model that may not be currently available. The acquisition of this model is explained in the next section. When more than one taxon is retrieved by the `plausible-descriptions` task, the `select-description` task uses the `user-preferences` method to choose only one taxon. Finally the `reflect-result` task completes the `solved episode` model of the new specimen by instantiating as value of the taxonomy level solution for the new example the obtained taxon. The `identify-by-subsumption` method fails if there is no taxon description subsuming the new example. Notice that a new sponge specimen can be incompletely identified, i.e. it could have only been classified into some taxonomy levels.

2.2.2. The `KA-Task`

The goal of the KA-Task is to acquire the descriptions for all taxa in order to build the `descriptions of taxa` model required by the `plausible-descriptions` task. The descriptions have to be learned using some inductive learning method. In particular, the KA-Task can use either INDIE or DISC to induce the taxa descriptions. As it has been explained in chapters 5 and 6 respectively, both INDIE and DISC obtain discriminant descriptions. In other words, a description for a taxon T_k subsumes all the examples belonging to T_k and does not subsume negative examples. Negative examples of a taxon T_k are all the specimens that belong to the rest of taxa in the taxonomic level of T_k. According to the results of INDIE and DISC, a taxon can be described by a disjunctive description.

In the next section we will analyse the description that SPIN obtains for the taxa using INDIE, DISC and LID.

3. Evaluation of SPIN

The case base contains descriptions of 26 sponges belonging to the family Geodiidae (Demospongiae:Astrophorida). Below the family Geodiidae there are 5 genus (*caminus, erylus, geodia, isops* and *pachymatisma*) and 11 species. Some of the specimens of the case base are completely described whereas some others have an incomplete description.

The specimens used by SPIN, even they are representatives of taxa to which they belong, they are not a good sample of the existing sponge populations. Consequently, it is not a dataset were prediction on unseen examples can be meaningfully. Figure 9.1 shows that some taxa of the level *species* have only one specimen (*erylus corsicus* and *erylus pachydermata*), and most of the species have only two or three specimens. In taxa of the *genus* level the situation is better since all have more than one example (may be incompletely described). Thus, the performance of SPIN in the `identification` task is evaluated at the *genus* level.

SPIN can be analysed under two points of view: according to the results of the identification (predictivity) and according to the description for each taxon learned using INDIE, DISC or LID. For the reasons expressed above, we have not evaluated the predictivity of SPIN because of the short size of the case base. Therefore only the descriptions obtained by the learning methods are explained in detail.

To evaluate the LID performance we need to identify some new specimen, so we have used the leave-one-out method technique. The application of this technique means that the identification of some specimens may be difficult for two reasons. One reason is that the specimens are highly representative of each taxon, so the extraction of one of them means to loose a significant part of the taxon. A second reason is that taxa such as *caminus* and *pachymatisma* have few specimens (3 and 2 respectively), so there is a little sample to identify a new specimen.

In the next section we explain the descriptions obtained by SPIN in using INDIE, DISC and LID. Then, we present a summary of the discussion of these descriptions with an expert.

3.1.2. Analysis of the descriptions of taxa

Each taxon T can be described by a disjunctive description $D = \{D_k\}$ where for each sponge $S \in T$ such that there is a D_k subsuming S, and if $S \notin T$ then $\forall D_k \in D : D_k \not\sqsubseteq S$. In this section we present the descriptions obtained from the taxa at the *genus* level.

Genus Caminus

There are three specimens (namely E1, E2 and E3) belonging to genus *caminus* in the case base. Two of these specimens (E1 and E2) have a detailed description (around 12 features) whereas the other (E3) is described using only 3 features.

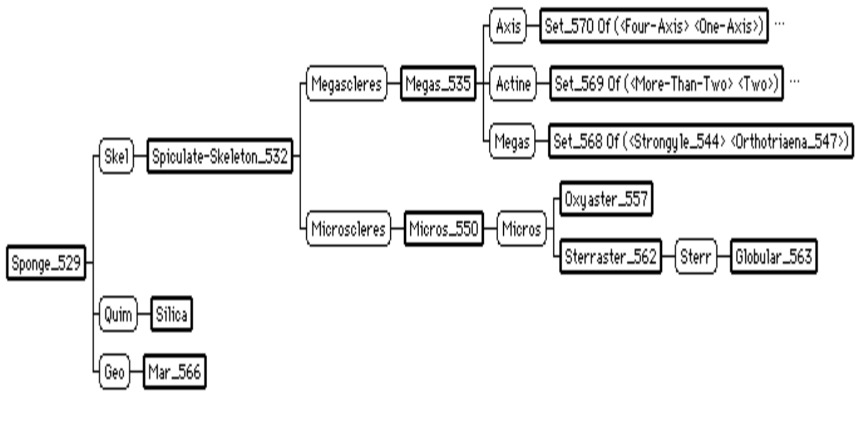


Figure 9.5. Description obtained from the anti-unification of the specimens belonging to the genus *caminus* in the case base of SPIN.

From the anti-unification of E1, E2 and E3 the feature term show in figure 9.5 is obtained. This description does not subsume negative examples, so it is a discriminant description for genus *caminus*. Applying the generalisation post-process, INDIE obtains the description in figure 9.6 for the genus *caminus*.

Let D = *any* be the current description for genus *caminus*. Since D subsumes negative examples (specimens belonging to other genus), DISC has to specialise it. The first step is to build a description d_a from the anti-unification of E1, E2 and E3 (description in figure 9.6). The bias of the DISC method selects as candidates to specialise D those features that are leaves of d_a. From this set, DISC selects the feature `megas` as the most discriminant. Finally, the path from the root of d_a to `megas` is added to D, obtaining the same description as INDIE for genus *caminus* (see figure 9.6).

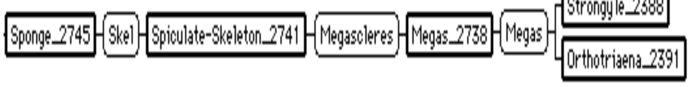


Figure 9.6. Description obtained by INDIE (with the generalisation post-process) and DISC for the *caminus* genus.

The use of LID to identify specimens has produced the following results. The specimens E1 and E3 have been correctly identified as belonging to the genus *caminus*. The description provided by LID to justify this identification is the same as that of INDIE and DISC (i.e. that in figure 9.6). While identifying the specimen E2, LID produces a multiple solution answer that includes the correct solution.

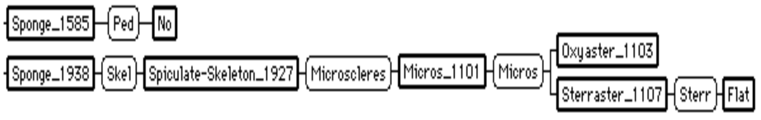


Figure 9.7. Disjunctive description obtained by INDIE to describe sponges belonging to the genus *erylus*.

Genus Erylus

Nine of the specimens (namely E4, E5 ... E12) in the case base belong to the genus *erylus*. Each specimen is usually described by about 6 features, but there are few features common to all them. First, INDIE obtains an initial hypothesis by anti-unification. Since it is not discriminant, INDIE specialises and it finds the disjunctive description (composed by two disjuncts) in figure 9.7.

The heuristic of DISC selects the `sterr` feature as most discriminant. This feature can take two values: *sterr* and *flat*. Therefore, the specimens of the genus *erylus* are partitioned according to these values obtaining the partition ((E4, E5, E6, E7, E8, E10, E11, E12) (E9)), where the `sterr` feature takes in E9 the value *globular*, and in the remaining specimens takes the value *flat*. The description D1 build with the `sterr` feature having value *flat* is discriminant but the description D2 containing the `sterr` feature having value *globular* is not. So, D2 has to be specialised by selecting the next most discriminant feature. After this step, DISC finds that the next feature to use for specialisation is `ped`. Finally, the disjunctive description that characterises the sponges belonging to the genus *erylus* is that of figure 9.8

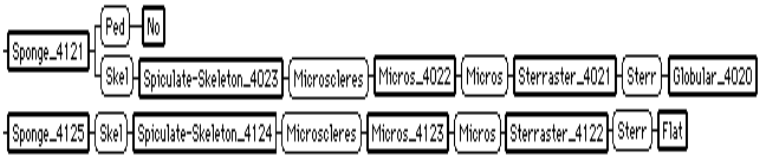


Figure 9.8. Disjunctive description obtained by DISC for the genus *erylus*.

There are two remarks to make about these results. On the one hand, the `sterr` feature with value *flat* is characteristic of the sponges belonging to the genus *erylus*. On the other hand, the description of E9 is likely to be erroneous and also human experts fail in their identification (Domingo, personal communication).

The LID results using the leave-one-out technique are the following: 3 correct answers, 4 multiple solution answers including the correct solution, 1 multiple solution answer without including the correct solution and 1 incorrect answer. For the correct answers, LID has provided the description in figure 9.9 as justification of the membership to the genus *erylus*.

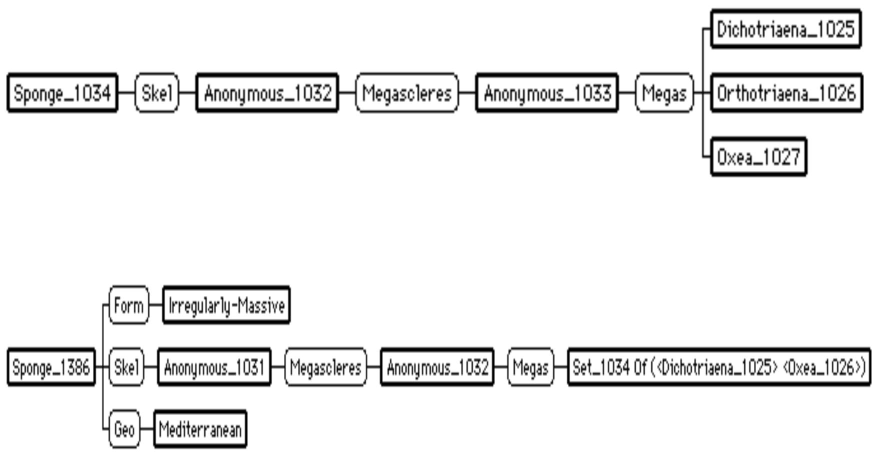


Figure 9.9. Justification provided by LID for the specimens belonging to the genus *erylus*.

We have not applied the majority rule to the multiple solution answers, since most of them produce an incorrect answer[3]. In fact, LID identifies these specimens as belonging to the genus *isops*. The genus *erylus* and *isops* are very similar. The main difference between them is that the genus *erylus* is characterised by a flat sterr in the microscleres whereas the genus *isops* has not a flat sterr (Domingo, personal communication). LID does not select the `sterr` feature as the most relevant since this feature is not present in all the training sponges. therefore it does not appear in the built description. As a consequence, without this feature it is very difficult to distinguish both genus.

[3] This is quite logical since, as we stated before, the sponges dataset is not a good sample of sponges population, and the majority rule only works if we assume a good population sample in the dataset (so that most frequent in the sample implies more frequent in the real population).

Genus Geodia

The case base of SPIN has 8 sponges (namely E13 ... E20) belonging to the genus *geodia*. These sponges are described by means of a number of features varying from 5 to 10. This variability among the descriptions makes it difficulty to find a set of discriminant features common to all the *geodia* specimens. So, INDIE needs several specialisations of the description obtained from the anti-unification of E13 ... E20 in order to build a discriminant description D. Figure 9.10 shows the disjunctive description (composed of 5 disjuncts) obtained by INDIE to describe the genus *geodia*.

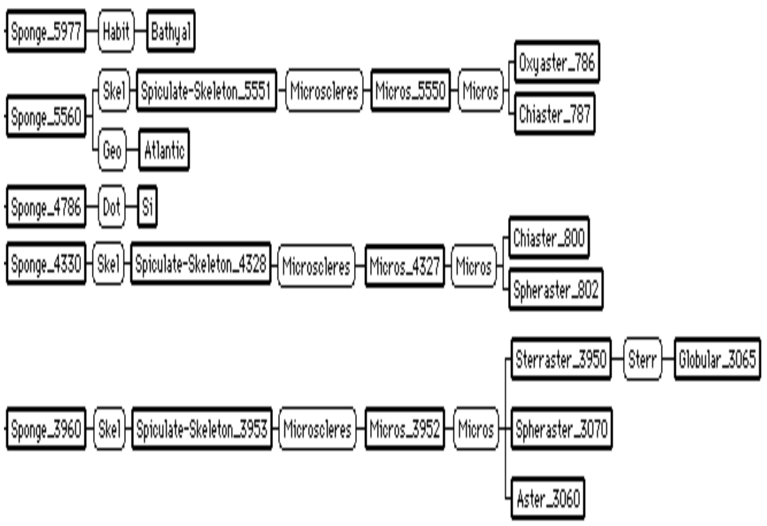


Figure 9.10. Disjunctive description for the genus *geodia* obtained by INDIE

The variability of the sponge descriptions affects the DISC results in constructing the set of candidate features to specialise a current description. In other words, there are few features common to E13 ... E20, therefore the set of candidates to specialise the current description is also reduced. As a consequence, it may be necessary to select several common features in order to obtain a discriminant description. Notice that for each selected feature several disjuncts have been introduced (one for each different value of the selected feature). Figure 9.11 shows the disjunctive description (composed of 6 disjuncts) obtained by DISC.

Concerning LID, the results obtained have been the following: 2 correct answers (specimens E15 and E18), 4 multiple solution answers including the correct solution(E13, E16, E19 and E20), 1 multiple solution answer without the correct solution and 1 incorrect solution. The justification for the membership of E15 and E18 to the genus *geodia* is shown in figure 9.12.

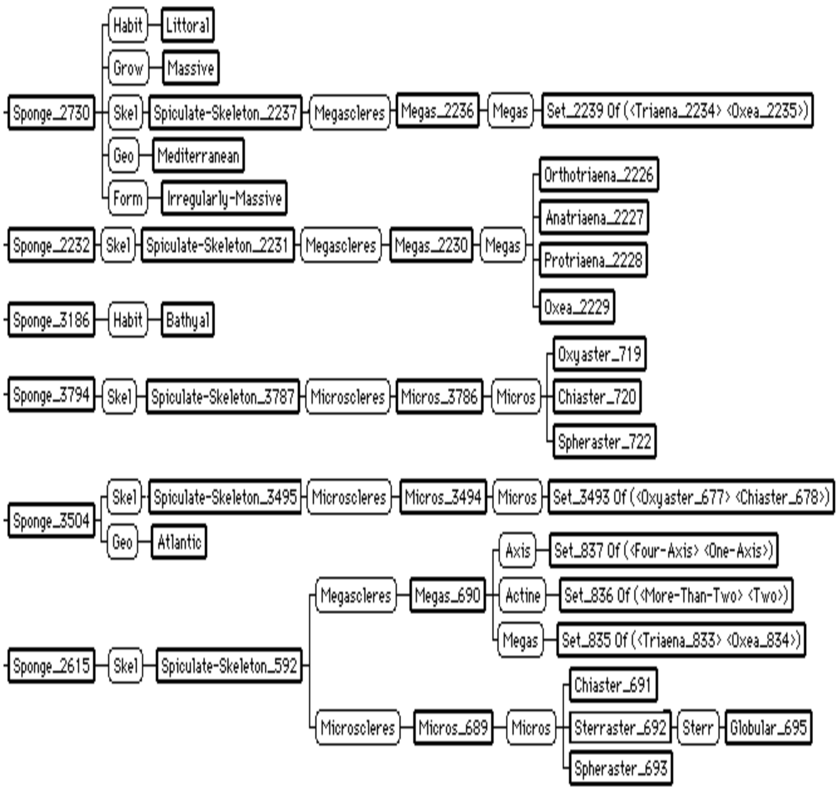


Figure 9.11. Disjunctive description for the genus *geodia* obtained by DISC.

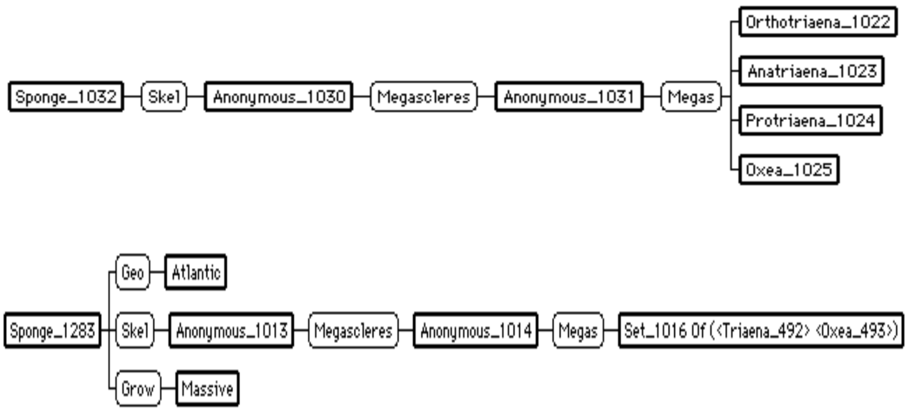


Figure 9.12. Descriptions build by LID to justify that E15 and E18 belong to the genus *geodia*.

Genus Isops

The case base of SPIN contains four sponges (namely E21, E22, E23 and E24) described using respectively 11, 5, 13 and 12 features. The description D obtained from their anti-unification has 5 common features (`skel`, `quim`, `form`, `grow` and `geo`). According to INDIE, D has to be specialised since it subsumes some negative examples. As in the genus above, INDIE needs several specialisation steps to find the discriminant disjunctive description in figure 9.13.

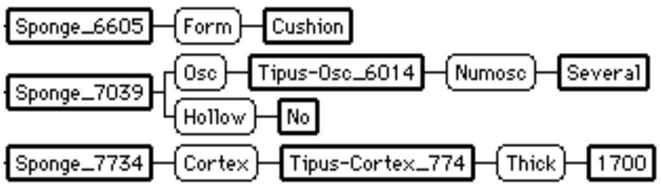


Figure 9.13. Disjunctive description obtained by INDIE for the genus *isops*.

Concerning DISC, the heuristic selects `sterr` as the most relevant feature. In the specimens of the genus *isops*, the `sterr` takes only the value *globular*, so more specialisations are necessary. Figure 9.14 shows the disjunctive description obtained by DISC for the genus *isops*.

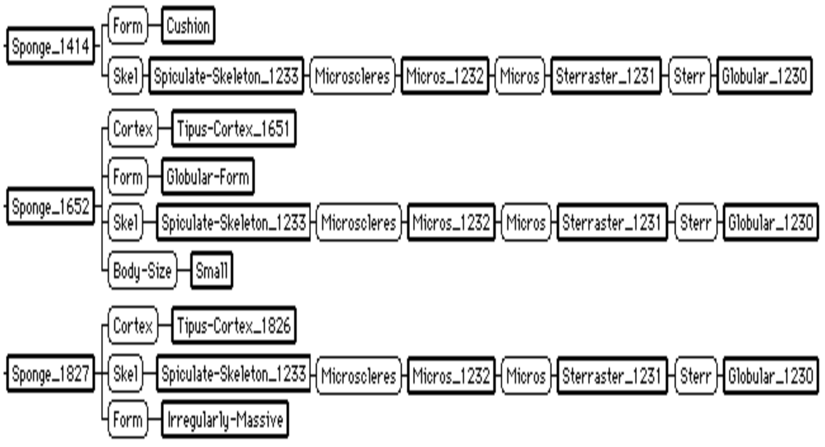


Figure 9.14. Disjunctive description obtained by DISC for the genus *isops*.

LID provides multiple solution answers in identifying specimens belonging to the genus *isops*. Only one of these answers does not include the correct solution (the identification of E21).

Genus Pachymatisma

There are only two specimen (E25 and E26) in the case base that belong to the genus *pachymatisma*. Both specimen have very similar descriptions. In fact they only differ in the description of the *spiculate-skeleton* feature term as value of the `skel` feature. The *spiculate-skeleton* in E25 has a feature called `spicarch` whereas this feature is not present in E26.

The description D obtained from the anti-unification of E25 and E26 has the same features and values as the description of E26. This description D does not subsume negative examples, so INDIE (after the post-process) produces as result the description in figure 9.15.

Concerning DISC, it uses the description D as bias to select a feature to specialise the current description (initially *any*). The feature selected as the most discriminant is `pach`, so DISC produces the same discriminant description as INDIE.

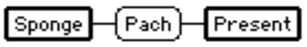


Figure 9.15. Description obtained by INDIE and DISC for the genus *pachymatisma*.

LID identifies E25 and E26 as belonging to the genus *pachymatisma* or to the genus *caminus*. Let us analyse this result. LID builds a description in a way similar to DISC. The main difference is that DISC only uses the examples belonging to the current class whereas LID uses the complete training set. As a consequence, features common to all the examples of a class may not be common to all the examples. So, DISC biases these features to specialise the current description whereas LID does not.

In particular, specimen E25 and E26 have many common features and DISC takes the `pach` feature as the most relevant. In fact, this feature is very characteristic for the genus *pachymatisma* (Domingo, personal communication). Nevertheless, the `pach` feature is not common to all the training examples, so it does not appear in the description build by LID (figure 9.16).

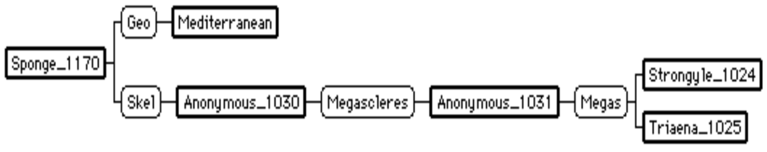


Figure 9.16. Description build by LID to identify the specimens E25 and E26.

3.1.3. Discussion

The descriptions obtained using INDIE and DISC have been presented to a domain expert. In general, the expert has considered these descriptions accurate. This is because although 26 specimen are few they are, however quite representative of the taxa they belong.

Most of the descriptions obtained are different from the descriptions that an expert would provide. For instance the description provided by INDIE and DISC for genus *pachymatisma* (figure 9.15) is very short (it contains only the feature `pach`). Our expert would provide descriptions having more features but she also thinks that the descriptions obtained by INDIE (with the post-process) and DISC indeed contain those features in which an expert focuses his attention to classify a specimen (Domingo, personal communication).

Concerning LID, applied over the domain of marine sponges, it provides a high number of answers with multiple solutions (most of them including the correct one). LID results cannot be compared to those of INDIE and DISC since the descriptions built by the inductive methods have used all the available specimens. Instead, LID (that is a lazy learning method) has been evaluated using the leave-one-out technique. That is to say, one example perhaps very representative of a taxon, has been leaven for identification. As a consequence, due to the narrowing strategy followed by LID over such a small dataset, results of the identification are not satisfactory. The use of the majority rule is not appropriate because of the size of the dataset.

4. Conclusions

The goal of SPIN was to solve the identification task in the domain of marine sponges. The identification task in SPIN can be solved using two methods: the `explore-taxonomy` and the lazy learning method (LID) explained in chapter 7. The `explore-taxonomy` method uses the descriptions of the taxa to identify a new specimen (sponge). These descriptions are inductively learned from the available specimens using DISC or INDIE.

Because the available sponges are highly representative of the taxa they belong, the obtained descriptions are accurate. Also the descriptions obtained by INDIE, DISC and LID are often equivalent. The opinion of a domain expert is that the descriptions inductively built by the inductive methods (INDIE and DISC) contain features that characterise the taxa. These descriptions are very different (usually shorter) than those found in the literature.

As future work we plan to extend the case base in order to obtain descriptions for other others and families of the *demospongiae* class. This extension will be made in collaboration with experts in marine sponges of the CEAB-CSIC institute.

From the obtained results by extending the case base, it will be interesting to compare the results of SPIN with those produced by the SPONGIA Expert System in identifying the same specimens. Also we can use all the information reported in the literature about the available specimens since we have used in SPIN the same features that SPONGIA used.

Chapter 10

Conclusions and Future Work

This summary chapter contains a short list of the contributions of this thesis and outlines of some issues which can be object of future research.

1. Contributions

The goal of this thesis is to integrate learning and problem solving. We have chosen Knowledge Modelling (KM) methodologies to make this integration since they are methodologies allowing the construction of KS with problem solving capabilities. We have defined a framework for developing knowledge systems where learning capabilities are integrated to the problem solving process resulting from a KM analysis of the application domain.

Moreover, elements of our framework can be represented using the feature terms formalism. This formalism solves the question of the different representations used by both Knowledge Modelling and Machine Learning techniques.

Summarising, the contributions of the framework we propose are the following:

- *Use of Knowledge Modelling methodologies to analyse Machine Learning techniques as methods.* As a consequence, learning methods are analysed by means of a task/method decomposition and the models that a learning method requires and constructs for each task. In this view, a learning method is like a problem solving method.
- *Uniform representation of problem solving and learning methods.* Learning methods are associated to tasks, called KA-Tasks, whose goal is the acquisition of knowledge required from some problem solving

activity. Therefore, we propose that an application with learning capabilities, can be analysed in terms of tasks, models and (problem solving and learning) methods.

- *Problem solving and learning integration.* The KM analysis of a domain application with learning capabilities produces a task/method decomposition. During the Design phase, the acquisition of some models is delayed to the Problem Solving phase. For each of these models a KA-Task and its associated learning method have to be defined. Therefore, the task/method decomposition representing a domain application includes tasks (solved using PSM) and KA-tasks (solved using learning methods).

- *Lazy problem-centred selection of PSM during the Problem Solving phase.* In our framework, a task (or KA-Task) can have more than one PSM associated. The selection of the appropriate PSM can be delayed until the Problem Solving phase, being the implemented system the responsible of this selection. In other words, we propose to select the appropriate method for a task during the problem solving phase according to the information available when the problem is being solved. As a consequence, the knowledge required to make this selection has to be acquired during the KM analysis of the application domain.

- *Multistrategy Learning Systems.* A Multistrategy Learning system is a specific combination of learning methods. During the KM analysis of a domain, several KA-Tasks can be identified and each KA-Task can be achieved using a specific learning method. In our framework, the integration of multiple learning methods is domain specific (since it depends on the KM analysis) and has not a pre-defined strategy (since it depends on the PSM selected to solve each new problem). In other words, when the PSM selected to solve a task requires a not yet available model, its KA-Task (with an associated learning method) will be used to acquire it. Therefore, the learning performed is adapted to both the domain task and the current task environment. Moreover, each KA-Task, as any task, may have associated more than one learning methods if need be.

The integration of Machine Learning and Knowledge Modelling needs to solve a practical issue: the different representation formalisms used by both techniques. We have addressed this issue using the NOOS representation language. NOOS uses feature terms (that are a generalisation of first-order terms) as representation formalism. As a consequence, we need to introduce learning methods capable to deal with feature terms. In Part II we have introduced three learning methods whose contributions are the following:

- *Learning using feature terms.* We have shown that relational learning using feature terms is feasible. Moreover, the learning methods we propose can deal with relational datasets commonly used in relational learning and ILP.

- *Learning as construction of models.* Problem solving is viewed as a process of construction of models. A PSM solves a problem by building the `solved episode` model of that problem. An inductive learning method is a process that, from past problems (`solved episode` models) constructs a model M (required by a PSM). Then, this model M is used by the PSM to build a `solved episode` model for the current problem. A lazy learning method directly builds a new `solved episode` model from past problems (`solved episode` models).

- *Inductive learning methods using feature terms.* Feature terms form a partial order with respect to the subsumption relation. In this context, induction can be seen as a search process in the space of feature terms. We have defined two inductive learning methods handling feature terms: INDIE and DISC. INDIE follows a heuristic bottom-up strategy to build a concept description whereas DISC follows a heuristic top-down strategy.

- *Lazy learning methods using feature terms.* Feature terms allow the structured representation of the examples. We have proposed a lazy-learning method, LID, for CBR systems where cases have a structured representation.

- *Anti-unification for biasing learning methods.* INDIE is a bottom-up method that uses anti-unification as basis to build concept descriptions. DISC is a top-down method that uses the anti-unification to select the features candidates to specialise a current description. LID uses anti-unification similarly to DISC.

Using the framework proposed in Part I we have analysed both application domains (Chromatography and Marine Sponges). The obtained analysis and the learning methods introduced in Part II have been used to develop two domain applications: CHROMA and SPIN. CHROMA is a tool supporting protein purification. SPIN is an application to the identification of marine sponges. CHROMA shows the integration of learning and problem solving, and also the capability of lazy problem-centred selection of methods. SPIN integrates learning and problem solving and uses all the learning methods proposed in Part II.

2. Future Work

This work has shown that several issues are interesting enough for future research. Some of these issues are the following:

- *Analysis of other biases for the inductive learning methods.* The learning methods defined in Part II have a bias that considers only the features that appear as leaves of the description obtained from the anti-unification of the examples. In particular for DISC, it would be interesting to relax this bias, i.e. to consider any of the features included in the description obtained from the anti-unification.
- *Improvements of the methods in Part II.* The learning methods in Part II have no special mechanism to deal with noisy values. We think that the incorporation of threshold mechanisms like those used in propositional learners and in FOIL could be useful. Moreover, ILP systems use biases controlling the number of predicates and variables that may be included into an inductive hypothesis. An equivalent bias mechanism over feature terms can be introduced by controlling the number of features and nodes of an inductive hypothesis (description).
- *Inductive learning of programs.* ILP systems use Horn clauses as representation formalism. The performance of Horn clauses has been widely studied in two fields: concept learning and program learning. We have analysed the feature terms performance in concept learning. Nevertheless, feature terms are used in NOOS to also represent methods. Therefore, in the future we plan to analyse how methods (i.e. programs) can be learned using feature terms.
- *Variants of the LID method.* A variant of LID is to store and reuse the description inductively build. This LID modification can be seen as an intermediate strategy between eager (inductive) and lazy learning. Another possible variant of LID is to use some domain knowledge (such as relevant features of the problem to be solved) to build the description from which the discriminant base is obtained.
- *Translation of propositional datasets.* Feature terms allow the structured representation of objects. Nevertheless, we have not used all the power of this formalism since examples of standard propositional datasets used in Part II have not been represented in a structured way. The reason is that we have not the domain knowledge required to translate them, in a meaningful way, to a structured representation. Only after having this translation, in addition to the mechanism to control noise, it will be worthwhile to perform further evaluations of INDIE, DISC and LID with other propositional datasets.

- *Improvement of CHROMA*. The selection of a PSM to solve the `purification` task is based on the `applicability conditions` model and on the `preferences` model, both acquired during the KM analysis of the domain. In the future we plan to analyse how to learn, for each PSM, the `applicability conditions` model and the `preferences` model. As a consequence, the acquisition of both models during the KM analysis was not necessary.

- *Improvement of SPIN.* The SPIN application can be improved by enlarging the case base in order to include other classes, families and orders of marine sponges. This improvement will be made in collaboration with spongiologists of the CEAB-CSIC.

APPENDIX A

Description of the Datasets

1. Robots Dataset

This dataset is used in (Lavrac and Dzeroski, 1994) to describe the LINUS system. Examples are tuples of attribute values labelled with a concept name. Domain objects are robots (see figure A.1) described by five features: `smiling`, `has-tie`, `holding`, `body-shape` and `head-shape`.

	Attributes and Values					
Robot	Class	Smiling	Holding	Has-tie	Head-shape	Body-shape
R1	friendly	yes	balloon	yes	square	square
R2	friendly	yes	flag	yes	octagon	octagon
R3	unfriendly	yes	sword	yes	round	octagon
R4	unfriendly	yes	sword	no	square	octagon
R5	unfriendly	no	sword	no	octagon	round
R6	unfriendly	no	flag	no	round	octagon

Figure A.1. Description of the robots used as inputs.

We have represented the robots (see figure A.2) as feature terms having two features: `description` and `solution`. The `description` feature contains the description of a robot, which is a feature term belonging to the sort *robot* and has the same five features as LINUS, i.e. `smiling`, `has-tie`, `holding`, `body-shape` and `head-shape`. The `solution` feature contains the class to which the described robot belongs, i.e. *friendly* or *unfriendly*.

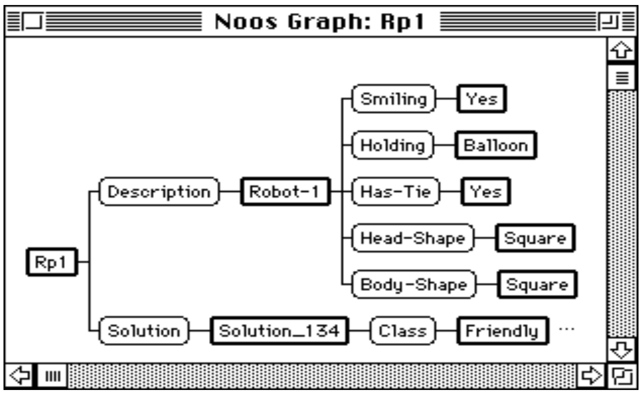


Figure A.2. A robot description using feature terms.

The hypotheses induced by LINUS have the form of if-then rules, for instance:

Class = friendly **if** [smiling = yes] $\wedge$ [holding = balloon]
Class = friendly **if** [smiling = yes] $\wedge$ [holding = flag]

Class = unfriendly **if** [smiling = no]
Class = unfriendly **if** [smiling = yes] $\wedge$ [holding = sword]

In section 4.1.1 of chapter 6 we have shown that DISC obtains similar descriptions as LINUS.

To obtain relational descriptions, LINUS needs to introduce background knowledge. This background knowledge is introduced by adding attributes whose value may be *true* or *false* according to the relation of other attributes.

For example, background knowledge can check for attributes with the same set of values). In the world of the robots this would lead to two new attributes that test the equalities [`smiling` = `has-tie`] and [`head-shape` = `body-shape`], named `same-shape`, the values are *true* and *false.* Using this idea the robot descriptions can be expressed in the following way:

	Attributes and Values						
Robot	Class	Smiling	Holding	Has-tie	Head-shape	Body-shape	Same-shape
R1	friendly	yes	balloon	yes	square	square	true
R2	friendly	yes	flag	yes	octagon	octagon	true
R3	unfriendly	yes	sword	yes	round	octagon	false
R4	unfriendly	yes	sword	no	square	octagon	false
R5	unfriendly	no	sword	no	octagon	round	false
R6	unfriendly	no	flag	no	round	octagon	false

Using the feature term representation described above, INDIE has obtained the same relational representation as LINUS using the additional attribute *same-shape* (see section 4.1.1 in chapter 5).

2. Drugs Dataset

The Drugs domain is used by the KLUSTER system (Kietz and Morik, 1994). Domain objects are descriptions of several drugs. Inputs to KLUSTER are the following assertions:

contains(aspirin, asa)	affects(oxazepun, stress)
contains(adumbran, coffein)	affects(phenazetin, headache)
contains(adumbran, oxazepun)	affects(asa, headache)
contains(anxiolit, oxazepun)	affects(prophymazon, headache)
contains(anxiolit, finalin)	sedative(adumbran)
contains(adolorin, phenazetin)	active(finalin)
contains(adolorin, prophymazon)	active(oxazepun)
contains(adolorin, nhc)	active(asa)
contains(placo, nhc)	active(phenazetin)
contains(alka-seltzer, nhc)	active(prophymazon)
contains(placo, sugar)	add-on(sugar)
contains(alka-seltzer, asa)	add-on(coffein)
excitement(stress)	add-odd(nhc)
pain(headache)	monodrug(alka-seltzer)
combidrug(anxiolit)	monodrug(aspirin)
combidrug(adolorin)	monodrug(adumbran)
placebo(placo)	anodyne(adolorin)
anodyne(aspirin)	anodyne(alka-seltzer)

From these descriptions KLUSTER builds a basic taxonomy, which is a hierarchy of primitive concepts and roles based on set inclusion between the known extensions of concepts and roles. From this taxonomy a set of learning problems is defined. The concepts that KLUSTER tries to define are taken top-down and breadth-first from the basic taxonomy. A concept learning problem of KLUSTER is to build discriminant definitions of mutually disjoint concepts (those having a same superconcept and that are mutually disjoint). A description is considered discriminant if the number of misclassified examples is lower or equal than a given threshold. KLUSTER obtains the following descriptions for the *monodrug* and *combidrug* concepts:

monodrug := drug and **atleast**(1, constains) and **atmost**(2, contains) and **atleast**(1, constains-active) and **at-most**(1, contains-active) and **atmost**(1, contains-add-on)

combidrug := drug and **atleast**(2, contains) and **atmost**(3, contains) and **atleast**(2, contains-active) and **atmost**(2, contains-active) and **atmost**(1, contains-add-on)

We have represented the drugs as feature terms. Thus, each drug (i.e. Alka-Seltzer, Aspirine, Adumbran, Adolorin and Anxiolit) is an object belonging to the sort *drug*. Objects in this sort contains two features: `contains` and `effects`. In turn, the feature `contains` has as values objects belonging to sort *active-substance*, to sort *add-on-substance*, or to both (since the value of the `contains` feature may be a set). Objects belonging to the *active-substance* sort can have the `affects` feature showing the symptom to which the active substance affects. The feature `effects` can take as value *anodyne* or *sedative*. The `solution` feature contains the solution class to which the described drug belongs. Figure A3 shows the background knowledge in the form of sorts relevant to the Drugs domain.

```
(define DRUG)

(define SYMPTOM)

(define (symptom STRESS))

(define (symptom HEADACHE))

(define (drug ANXIOLIT)
  (CONTAINS oxazepun finalin))

(define (drug ASPIRINE)
  (EFFECTS anodyne)
  (CONTAINS asa))

(define (drug ADUMBRAN)
  (EFFECTS sedative)
  (CONTAINS coffein oxazepun))

(define SET-EFFECTS)

(define (set-effects SEDATIVE))

(define (set-effects ANODYNE))

(define (drug ADOLORIN)
  (EFFECTS anodyne)
  (CONTAINS phenazetin prophymazon nhc))

(define (drug PLACO)
  (CONTAINS nhc sugar))

(define (drug ALKA-SELTZER)
  (EFFECTS anodyne)
  (CONTAINS asa nhc))
```

```
(define SUBSTANCE)

(define (substance ACTIVE-SUBSTANCE))

(define (substance ADD-ON-SUBSTANCE))

(define (add-on-substance COFFEIN))

(define (add-on-substance NHC))

(define (add-on-substance SUGAR))

(define (active-substance FINALIN))

(define (active-substance ASA)
  (AFFECTS headache))

(define (active-substance OXAZEPUN)
  (AFFECTS stress))

(define (active-substance PHENAZETIN))

(define (active-substance PROPHYMAZON))
```

Figure A.3. Representation of the background knowledge of the Drugs dataset using feature terms.

Thus, for instance DP6 has as value in the `description` feature, the feature term *Alka-Seltzer* belonging to the sort *drug*. This drug is a feature term having two features: `effects` (with value *anodyne*) and `contains` (with a set of values: *asa* and *Nhc*). The value *Asa* (*acetyl salicylic Acid*) is a feature term belonging to the sort *active-substance* and *Nhc* is a feature term belonging to the *add-on-substance* sort. The `solution` feature of DP6 has as value *monodrug*. In figure A.4 can be shown the description of the examples using feature terms.

```
(define (drug-problem DP1)            (define (drug-problem DP4)
 (DESCRIPTION anxiolit)                (DESCRIPTION adolorin)
 (SOLUTION (define (solution)          (SOLUTION (define (solution)
       (CLASS combidrug))))                  (CLASS combidrug))))

(define (drug-problem DP2)            (define (drug-problem DP5)
 (DESCRIPTION aspirine)                (DESCRIPTION placo)
 (SOLUTION (define (solution)          (SOLUTION (define (solution)
       (CLASS monodrug))))                   (CLASS placebo))))

(define (drug-problem DP3)            (define (drug-problem DP6)
 (DESCRIPTION adumbran)                (DESCRIPTION alka-seltzer)
 (SOLUTION (define (solution)          (SOLUTION (define (solution)
       (CLASS monodrug))))                   (CLASS monodrug))))
```

Figure A.4. Representation of the Drugs dataset using feature terms.

3. Arch Dataset

The Arch dataset was introduced by Winston (1975). This dataset has four examples of *figures* (figure A.5). A *figure* is composed of three *blocks*: two vertical and one horizontal. Two of the *figures* correspond to an arch figure and the remaining two *figures* are not arches.

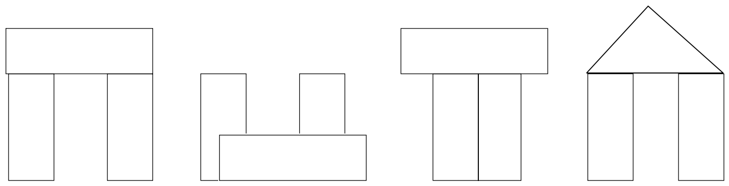


Figure A.5. Training examples of the Arch dataset

In the ILP framework, the goal is to induce the relation *arch*(A,B,C), stating that A, B, and C form an arch with columns B and C and lintel A. The following background relations were used: supports(X,Y), left-of(X,Y), touches(X,Y), brick(X), wedge(X) and parallelepiped(X).

The result of LINUS when the negative examples are explicitly provided is the following:

$$\text{arch(A,B,C)} \leftarrow \text{supports(B,A), not touches(B,C).}$$

i.e. A, B, and C form an arch if B supports A and B does not touch C. Regarding FOIL, it is not capable to obtain a description when the negative examples are explicitly given. When the negative examples are provided according the closed-world assumption FOIL and LINUS provide the following result:

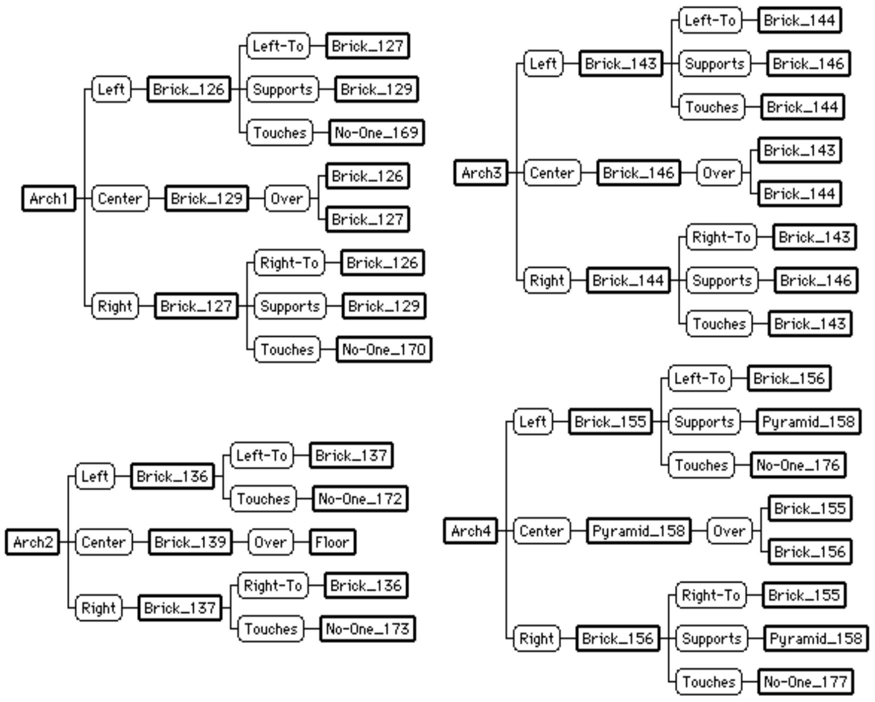
$$\text{arch(A,B,C)} \leftarrow \text{left-of(B,C), supports(B,A), not touches(B,C).}$$

i.e. A, B, and C form and arch if B is left to C and B supports A and B does not touch C. The definition obtained when the negative examples are explicitly given is more general than the obtained using the closed-world assumption.

As it has been explained in section 4.1.3 in chapter 5, the description obtained by FOIL and LINUS is consistent with the provided input examples nevertheless it covers some unseen negative examples. When the covered negative examples are also entered as training examples, LINUS obtains the following description:

$$\text{arch(A,B,C)} \leftarrow \text{supports(B,A), supports(C,A), not touches(B,C).}$$

Using feature terms we have represented the objects in figure A.5 in the following way:



Each figure is a feature term with root of the sort *figure* representing the descriptions above. A *figure* is described by three features: `left`, `center` and `right`. The values of these feature are feature terms belonging to the sort *brick*. *Brick* feature terms have four features: `left-to`, `right-to`, `supports` and `touches`. The `touches` feature of a *brick* B1 has as value the *brick* feature terms that are in contact with B1. If there are not *brick* feature terms touching B1, the value of the `touches` feature is *no-one*.

4. Families Dataset

The families dataset was introduced by Hinton (1989). This dataset contains the description of two families having twelve members each (figure A.6).

LINUS was used to learn the relation mother(A,B) from examples of this relation and the background relations father(X,Y), wife(X,Y), son(X,Y) and daughter(X,Y). Negative examples were generated under closed-world assumption.

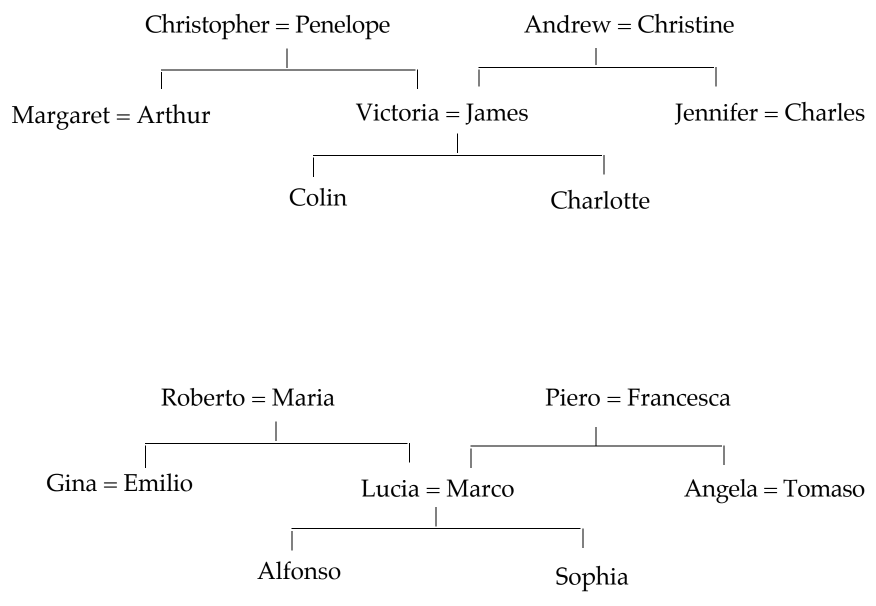


Figure A.6. Training set of the Families dataset.

Methods described in Part II search for concept descriptions instead of relations as LINUS. In particular, we have searched for the *mother* and *uncle* concepts (see section 4.1.4 in chapter 5 and section 4.1.4 in chapter 6). Negative examples of a concept are those *person* feature terms that do not belong to the concept. For instance, some negative examples of mother are Christopher, Andrew and Margaret (see figure A.6).

The representation of a family of this dataset using feature terms is shown in figure A.7. Each person of a family can belong to the sort *male* or *female* and its features are relations with other *persons* feature terms. So, for example, Charles belongs to the sort *male* and has three features: `wife`, `niece` and `nephew`. Instead, Colin also belongs to the sort *male* but they have 7 features: `mother`, `father`, `sister`, `grandfather`, `grandmother`, `uncle`, and `aunt`.

```
(define (male CHRISTOPHER)
  (SON Arthur)
  (DAUGHTER Victoria)
  (WIFE Penelope))

(define (female PENELOPE)
  (SON Arthur)
  (DAUGHTER Victoria)
  (HUSBAND Christopher))

(define (male ANDREW)
  (SON James)
  (DAUGHTER Jennifer)
  (WIFE Christine))

(define (female CHRISTINE)
  (SON James)
  (DAUGHTER Jennifer)
  (HUSBAND Andrew))

(define (female MARGARET)
  (HUSBAND Arthur)
  (NIECE Charlotte)
  (NEPHEW Colin))

(define (male ARTHUR)
  (WIFE Margaret)
  (FATHER Christopher)
  (MOTHER Penelope)
  (SISTER Victoria)
  (NIECE Charlotte)
  (NEPHEW Colin))

(define (male CHARLES)
  (WIFE Jennifer)
  (NIECE Charlotte)
  (NEPHEW Colin))

(define (female VICTORIA)
  (HUSBAND James)
  (FATHER Christopher)
  (MOTHER Penelope)
  (BROTHER Arthur)
  (SON Colin)
  (DAUGHTER Charlotte))

(define (male JAMES)
  (WIFE Victoria)
  (FATHER Andrew)
  (MOTHER Christine)
  (SISTER Jennifer)
  (SON Colin)
  (DAUGHTER Charlotte))

(define (female JENNIFER)
  (HUSBAND Charles)
  (FATHER Andrew)
  (MOTHER Christine)
  (BROTHER James)
  (NIECE Charlotte)
  (NEPHEW Colin))

(define (male COLIN)
  (FATHER James)
  (MOTHER Victoria)
  (SISTER Charlotte)
  (GRANDFATHER Christopher Andrew)
  (GRANDMOTHER Penelope Christine)
  (UNCLE Arthur Charles)
  (AUNT Jennifer Margaret))

(define (female CHARLOTTE)
  (FATHER James)
  (MOTHER Victoria)
  (BROTHER Colin)
  (GRANDFATHER Christopher Andrew)
  (GRANDMOTHER Penelope Christine)
  (UNCLE Arthur Charles)
  (AUNT Jennifer Margaret))
```

Figure A.7. Representation of a family using feature terms.

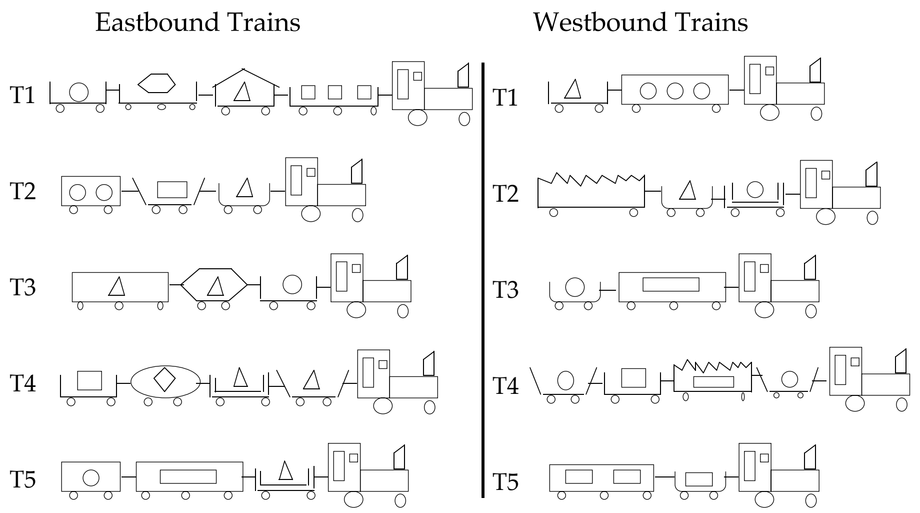


Figure A.8. Training examples of the Trains Dataset

5. Trains Dataset

This dataset was introduced by Michalski (1980) to test the INDUCE system. The Trains dataset is composed of the description of 10 trains having different numbers of cars with various shapes (figure A.8). The task is to distinguish between *eastbound* and *westbound* trains. Having structured objects with varying numbers of substructures, the task is difficult to be solved with attribute-value learning programs but should be appropriate for relational learning. Relations *eastbound(T)* and *westbound(T)* are defined in terms of the following relations:

has-car(T,C)	long(C)
open-rectangle(C)	u-shaped(C)
open-trapezoid(C)	ellipse(C)
closed-rectangle(C)	jagged-top(C)
sloping-top(C)	two-wheels(C)
three-wheels(C)	

Using ASSISTANT, LINUS generated a long description (consisting of 19 Prolog clauses) for the *eastbound* trains. After post-processing reduced to only one clause, identical to the one induced by FOIL.

FOIL also has been applied to this dataset. This system has no problem in finding the *eastbound* description. Nevertheless, FOIL is not capable to obtain descriptions covering all the positive examples of the *westbound* class. The INDUCE system is capable to obtain a description for the *westbound* class thanks to its ability to generate new descriptors.

The representation of this dataset using feature terms can be seen in figure A.9.

```
(define (train TRAIN-1)
 (WAGON1 (define (open-car)
         (WHEELS 2)
         (SIZE (define (long)))
         (FORM-CAR (define (openrect)))
         (LOAD-SET (define (rectanglod))
                 (define (rectanglod))
                 (define (rectanglod)))))
 (WAGON2 (define (closed-car)
         (WHEELS 2)
         (SIZE (define (short)))
         (FORM-CAR (define (slopetop)))
         (LOAD-SET (define (trianglod)))))
 (WAGON3 (define (open-car)
         (WHEELS 3)
         (SIZE (define (long)))
         (FORM-CAR (define (openrect)))
         (LOAD-SET (define (hexagonlod)))))
 (WAGON4 (define (open-car)
         (WHEELS 2)
         (SIZE (define (short)))
         (FORM-CAR (define (openrect)))
         (LOAD-SET (define (circlelod))))))

 (define (train TRAIN-6)
  (WAGON1 (define (closed-car)
          (WHEELS 2)
          (SIZE (define (long)))
          (FORM-CAR (define (closedrect)))
          (LOAD-SET (define (circlelod))
                  (define (circlelod))
                  (define (circlelod)))))
   (WAGON2 (define (open-car)
          (WHEELS 2)
          (SIZE (define (short)))
          (FORM-CAR (define (openrect)))
          (LOAD-SET (define (trianglod))))))
```

```
(define (train TRAIN-5)
 (WAGON1  (define (open-car)
          (WHEELS 2)
          (SIZE (define (short)))
          (FORM-CAR (define (dblopnrect)))
          (LOAD-SET (define (trianglod)))))
 (WAGON2 (define (closed-car)
         (WHEELS 3)
         (SIZE (define (long)))
         (FORM-CAR (define (closedrect)))
         (LOAD-SET (define (rectanglod)))))
 (WAGON3 (define (closed-car)
         (WHEELS 2)
         (SIZE (define (short)))
         (FORM-CAR (define (closedrect)))
         (LOAD-SET (define (circlelod))))))

 (define (train TRAIN-9)
  (WAGON1 (define (open-car)
          (WHEELS 2)
          (SIZE (define (short)))
          (FORM-CAR (define (opentrap)))
          (LOAD-SET (define (circlelod)))))
  (WAGON2 (define (closed-car)
          (WHEELS 2)
          (SIZE (define (long)))
          (FORM-CAR (define (jaggedtop)))
          (LOAD-SET (define (rectanglod)))))
  (WAGON3 (define (open-car)
          (WHEELS 2)
          (SIZE (define (short)))
          (FORM-CAR (define (openrect)))
          (LOAD-SET (define (rectanglod)))))
  (WAGON4 (define (open-car)
          (WHEELS 2)
          (SIZE (define (short)))
          (FORM-CAR (define (opentrap)))
          (LOAD-SET (define (circlelod))))))
```

Figure A.9. Some of the trains in figure A.7 represented as feature terms.

i.e. each train is a feature term belonging to the sort *train* which has a variable number of features. Thus train-1 has four `wagon` features whereas train-6 has two `wagon` features. Each `wagon` is a feature term that can belong to two sorts either *open-car* or *closed-car*. Both are subsorts of the *car-type* sort which features are the following: `wheels`, `size`, `form-car`, and `load-set`.

As we have seen in section 4.1.5 of chapter 5 and in section 4.1.5 of chapter 6, the representation of the trains dataset using feature terms allows the construction of descriptions for both *eastbound* and *westbound* classes.

6. Traffic Law Dataset

The Traffic Law dataset was introduced to test the MOBAL system (Morik et al., 1993) has background knowledge of some basic traffic regulations and a few sample cases of traffic violations. The Traffic Law domain is concerned with some basic knowledge about traffic regulations in Germany. The problem solving goal of the model is to derive a classification of the case along several dimensions, i.e. determining who will be held responsible for violation, how high the fine will be, and whether the responsible person will have to go to court.

The intention behind the creation of this domain model was to construct a model which could be used for experiments with MOBAL knowledge representation and learning modules. The datasets examples are about parking regulations, speed limits, vehicle safety and fines. The resulting domain, has 15 cases and background knowledge.

The basis for the Traffic Law domain model consists of a representation of several cases of traffic violations (see figure A.10). Each of these cases is represented by a number of MOBAL facts which provide featural and relational information.

The system automatically constructs a sort lattice based on the actual arguments used as predicate arguments in a domain model. Also this lattice can contain explicit symbolic sort definitions, provided by the user, for the predicates that are used (see (Morik et al., 1993) for more information about the MOBAL performance). MOBAL also has a predicate topology that either can be acquired automatically by the system based on the current set of rules or can be defined manually.

```
==============================
responsible(sw,event1).
not(appeals(sw,event1)).
not(court_citation(sw,event1)).
owner(sw,b_au_6773).
pays_fine(event1,sw).
involved_vehicle(event1,b_au_6773).
car_parked(event1,place1).
car_towed(event1,b_au_6773).
fine(event1,20).
bus_lane(place1).
sedan(b_au_6773).
not(tvr_points_p(event1)).

==============================
responsible(dj,event2).
appeals(dj,event2).
court_citation(dj,event2).
owner(dj,b_dx_1385).
involved_vehicle(event2,b_dx_1385).
car_parked(event2,place2).
sedan(b_dx_1385).
sidewalk(place2).
not(tvr_points_p(event2)).
not(unsafe_vehicle_violation(event2)).

==============================
responsible(mo,event3).
involved_vehicle(event3,hh_mo_195).
appeals(mo,event3).
car_parked(event3,place3).
court_citation(mo,event3).
owner(mo,hh_mo_195).
level_crossing(place3).
sedan(hh_mo_195).
not(tvr_points_p(event3)).

==============================
responsible(md,event4).
involved_vehicle(event4,b_md_4321).
appeals(md,event4).
car_parked(event4,place4).
court_citation(md,event4).
owner(md,b_md_4321).
fire_hydrant(place4).
sedan(b_md_4321).
not(tvr_points_p(event4)).

==============================
responsible(ab,event6).
involved_vehicle(event6,b_ab_89).
eco_expired(b_ab_89).
owner(ab,b_ab_89).
not(lights_necessary(event6)).
not(parking_violation(event6)).
sedan(b_ab_89).
not(tvr_points_p(event6)).

==============================
responsible(bc,event7).
involved_vehicle(event7,b_bc_90).
color(b_bc_90,blue).
eco_expired(b_bc_90).
owner(bc,b_bc_90).
sedan(b_bc_90).
not(tvr_points_p(event7)).

==============================
responsible(cd,event8).
time(event8,time8).
involved_vehicle(event8,b_cd_01).
dark(time8).
owner(cd,b_cd_01).
not(headlights_on(b_cd_01,event8)).
sedan(b_cd_01).
not(tvr_points_p(event8)).

==============================
responsible(de,event9).
involved_vehicle(event9,b_de_12).
appeals(de,event9).
court_citation(de,event9).
owner(de,b_de_12).
sedan(b_de_12).

==============================
responsible(ef,event10).
time(event10,time10).
involved_vehicle(event10,b_ef_23).
appeals(ef,event10).
court_citation(ef,event10).
owner(ef,b_ef_23).
faulty_brakes(b_ef_23).
fog(time10,place10).
not(headlights_on(b_ef_23,event10)).
location(event10,place10).
sedan(b_ef_23).
not(tvr_points_p(event10)).
```

Figure A.10. Some of the cases used by MOBAL.

Finally, the Traffic Law dataset also contains background knowledge of inferential relations in the domain. In figure A.11 there are the rules used as background knowledge.

sidewalk(X) $\rightarrow$ no-parking(X)
second-row (X) $\rightarrow$ no-parking(X)
bus-lane $\rightarrow$ no-parking(X)
fire-hydrant (X) $\rightarrow$ no-parking(X)
level-crossing $\rightarrow$ no-parking(X)
road-edge(X) $\rightarrow$ not(second-row(X))
road-edge(X) & not(no-parking-sign(X)) $\rightarrow$ parking-allowed(X)
car-parked(X,Y) & no-parking(Y) $\rightarrow$ parking-violation(X)
time(X,Y) & dark(Y) $\rightarrow$ lights-necessary(X)
time(O;X) & place(O,Y) & fog(X,Y) $\rightarrow$ lights-necessary(O)
involved-vehicle(X,Y) & major-corrosion(Y) $\rightarrow$ unsafe-vehicle-violation(X)
involved-vehicle(X,Y) & faulty-brakes(Y) $\rightarrow$ unsafe-vehicle-violation(X)
involved-vehicle(X,Y) & worn-tires(Y) $\rightarrow$ unsafe-vehicle-violation(X)

Figure A.11. Background knowledge used by MOBAL.

Authors first entered data consisting of a group of twelve sample cases so as to provide as basis for the learning process. Then the module of MOBAL called RDT was applied to a number of interesting predicates and several rules were discovered. They have found that some of the rules were not what they had intended, and augmented the knowledge base with additional facts and predicates. The RDT module finally arrived to the following set of rules:

R51: responsible(X,Y) & unsafe-vehicle-violation(Y) $\rightarrow$ appeals(X,Y)
R52: responsible(X,Y) & unsafe-vehicle-violation(Y) $\rightarrow$ court-citation(X,Y)
R53: court-citation(X,Y) & parking-violation(Y) $\rightarrow$ appeals(X,Y)
R54: involved-vehicle(X,Y) & owner(Z,Y) $\rightarrow$ responsible(Z,X)
R55: court-citation(X,Y) & unsafe-vehicle-violation(Y) $\rightarrow$ appeals(X,Y)
R56: appeals(X,Y) & parking-violation(Y) $\rightarrow$ court-citation(X,Y)
R57: appeals(X,Y) & unsafe-vehicle-violation(Y) $\rightarrow$ court-citation(X,Y)
R58: involved-vehicle(X,Y) & not(buckled-up(X,Y)) $\rightarrow$ not(tvr-points-p(X))
R59: involved-vehicle(X,Y) & lights-necessary(X) & not(headlights-on(Y,X)) $\rightarrow$ not(tvr-points-p(X))
R60: parking-violation(X) $\rightarrow$ not(tvr-points-p(X))
R61: lights-necessary(X) $\rightarrow$ not(tvr-points-p(X))

In section 1.2 of chapter 4 we have described how the background knowledge is represented using feature terms. An example of how the cases in figure A.10 are represented using feature terms is the following:

```
(define (traffic-law-case tlc-1)
 (DESCRIPTION (define (description-case)
          (event (define (event)
               (involved-vehicle (define (vehicle)
                          (owner SW)
                          (sedan true)))
               (car-parked (define (bus-lane)))
               (car-towed (>> involved-vehicle event))
               (responsible SW)
               (fine 20)
               (pays-fine SW)))
          (appeals false)
          (tvr-points-p false)))
 (SOLUTION no-court-citation))
```

In particular, the TLC-1 is a feature term belonging to the sort *traffic-law-case* that has two features: `description` and `solution`. The `solution` feature indicates the kind of traffic violation that describes the case. In section 4.1.6 of chapter 5 and in section 4.1.6 of chapter 6 can be seen the results of the application of INDIE and DISC respectively to this dataset.

7. Mesh Dataset

The finite element method is frequently used to analyse stresses in physical structures represented quantitatively as finite collections (meshes) of elements. Engineers partition the structure into a finite number of connected elements and the deformation of each element is computed using linear algebraic equations. In order to design a numerical model of a physical structure it is necessary to decide the appropriate resolution for modelling each component part.

The mesh would represent the exact shape of the structure. Fine meshes are adequate where the expected deformations are small. Ideally, the coarsest mesh which gives rise to sufficiently low errors is employed. This minimises the required computation time since each additional element adds an extra linear algebraic equation to the set which must be solved.

Too fine a mesh leads to unnecessary computational overheads when executing the model, while too coarse a mesh produces intolerable approximation errors. It is very difficult to known in advance where the mesh should be fine and where it should be coarse because a number of parameters have to be considered (i.e. shape of the structure, loadings and boundary conditions).

Usually it is necessary to make a few different meshes until the right one is found. The trouble is that each mesh must be analysed, since we generate the next mesh on the base of the results derived from the previous mesh. Each mesh analysis can take several days of computer time, so iterative analysis can be very costly. There exists a great need for knowledge-based systems which are able to automatically design finite element meshes.

An attribute-value representation used in most of the available Machine Learning systems is essentially inappropriate for representing this problem since a reasonable representation of the geometry of a structure must include the relations between its primitive components, which cannot be represented naturally in an attribute-value language. Instead, ILP techniques have been applied. Dolsak applied the ILP system called GOLEM (Dolsak and Muggleton, 1992) to construct rules deciding on appropriate mesh resolution. Then the same problem was used in FOIL and mFOIL.

The resolution of a finite element mesh is determined by the number of elements on each of its edges. The problem of learning rules for determining the resolution of a finite element mesh can be formulated as a problem of learning rules to determine the number of elements on an edge. The training examples have the form *mesh*(E,N) where E is an edge label (unique for each edge) and N is the number of elements on the edge denoted by label E. N takes values from 1 to 17.

In preliminary experiments (Dolsak and Muggleton, 1992), three objects (structures) with their corresponding meshes were used in the learning process: a hook, a hydraulic press cylinder and a paper mill. In figure A.12 there is the representation of the hook used in GOLEM.

```
mesh(b1,9)      mesh(b8,9)
mesh(b2,1)      mesh(b9,1)
mesh(b3,2)      mesh(b10,2)
mesh(b4,7)      mesh(b11,7)
mesh(b5,1)      mesh(b12,1)
mesh(b6,1)      mesh(b13,1)
mesh(b7,1)      mesh(b14,1)
```

Figure A.12. Meshes corresponding to the hook structure used in GOLEM.

FOIL and LINUS gathered data about three additional structures: a pipe connector, a roller and a bearing box. Our experiments have used the same structures that GOLEM, since we have not an exhaustive information about the results obtained by LINUS and FOIL.

- Types of the edges

important-long(b1)
important-long(b8)

important(b4)
important(b11)

important-short(b3)
important-short(b10)

not-important(b2)
not-important(b5)
not-important(b6)
not-important(b7)
not-important(b9)
not-important(b12)
not-important(b13)
not-important(b14)

- Boundary conditions

free(b2)
free(b3)
free(b7)
free(b9)
free(b10)
free(b14)

one-side-fixed(b1)
one-side-fixed(b4)
one-side-fixed(b8)
one-side-fixed(b11)

two-side-fixed(b5)
two-side-fixed(b6)
two-side-fixed(b12)
two-side-fixed(b13)

- Loads

not-loaded(b1)
not-loaded(b2)
not-loaded(b5)
not-loaded(b6)
not-loaded(b7)
not-loaded(b8)
not-loaded(b9)
not-loaded(b12)
not-loaded(b13)
not-loaded(b14)

one-side-loaded(b3)
one-side-loaded(b4)
one-side-loaded(b10)
one-side-loaded(b11)

- Geometric representation

neigbour-xy-r(b1,b13)
neigbour-xy-r(b13,b8)
neigbour-xy-r(b8,b7)
neigbour-xy-r(b7,b1)
neigbour-xy-r(b4,b6)
neigbour-xy-r(b6,b11)
neigbour-xy-r(b10,b14)
neigbour-xy-r(b14,b3)

neighbour-zx-r(b5,b1)
neighbour-zx-r(b8,b9)
neighbour-zx-r(b9,b10)
neighbour-zx-r(b10,b11)
neighbour-zx-r(b11,b12)
neighbour-zx-r(b12,b8)
neighbour-zx-r(b1,b2)
neighbour-zx-r(b2,b3)
neighbour-zx-r(b3,b4)
neighbour-zx-r(b4,b5)

neighbour-yz-r(b5,b6)
neighbour-yz-r(b6,b12)
neighbour-yz-r(b12,b13)
neighbour-yz-r(b13,b5)
neighbour-yz-r(b2,b7)
neighbour-yz-r(b7,b9)
neighbour-yz-r(b9,b14)
neighbour-yz-r(b14,b2)

opposite-r(b11,b8)
opposite-r(b1,b8)
opposite-r(b3,b1)
opposite-r(b3,b1)
opposite-r(b4,b1)
opposite-r(b10,b8)

same-r(b1,b8)

Figure A.13. Description of the edges composing the hook structure used by the GOLEM system. Some predicates such as `neighbour-xy-l`, `neighbour-yz-l`, `neighbour-zx-l`, `opposite-l` and `same-l` are not show here. They have the same arguments as the predicates with suffix -r.

GOLEM needs three types of input data to build the rules:

- Foreground examples that contains the examples of classified edges of meshes (as in figure A.12).

- Negative examples that contain all possible combinations of names and number other than those found in the foreground examples.
- Background facts that are definitions of the vocabulary that can be used to describe hypotheses about meshes. The contents of the background are divided into five parts: declarations, type of edges, boundary conditions, loads, and geometric representations. Figure A.13 shows the definitions of the elements of the hook.

Using feature terms we have defined two main sorts: *mesh* and *edge*. Objects of the *mesh* sort have two features: `description` and `solution`. The `description` feature contains the description of an object of sort *edge*. The `solution` feature contains the solution class to which the described edge belongs. Some of the meshes and edges of the hook structure are the following:

```
(define (edge B1)                  (define (mesh-problem M349)
  (TYPE half-circuit-hole)           (DESCRIPTION b1)
  (BOUNDARY-CONDITIONS fixed)        (SOLUTION nine))
  (LOADINGS no-loaded)
  (NEIGHBOUR-XY-R B34)
  (NEIGHBOUR-YZ-R B2)              (define (mesh-problem M22)
  (NEIGHBOUR-XY-L B34)               (DESCRIPTION b2)
  (NEIGHBOUR-YZ-L B2)                (SOLUTION one))
  (OPPOSITE-R B3)
  (OPPOSITE-L B3))
                                   (define (mesh-problem M24)
(define (edge B2)                    (DESCRIPTION b3)
  (TYPE not-important)               (SOLUTION two))
  (BOUNDARY-CONDITIONS fixed)
  (LOADINGS no-loaded)
  (NEIGHBOUR-YZ-R B3)              (define (mesh-problem M25)
  (NEIGHBOUR-ZX-R B5)                (DESCRIPTION b4)
  (NEIGHBOUR-YZ-L B3)                (SOLUTION seven))
  (NEIGHBOUR-ZX-L B5))

(define (edge B3)                  (define (mesh-problem M26)
  (TYPE half-circuit)                (DESCRIPTION b5)
  (BOUNDARY-CONDITIONS fixed)        (SOLUTION one))
  (LOADINGS no-loaded)
  (NEIGHBOUR-XY-R B5)
  (NEIGHBOUR-YZ-R B4)              (define (mesh-problem M27)
  (NEIGHBOUR-XY-L B5)                (DESCRIPTION b6)
  (NEIGHBOUR-YZ-L B4))               (SOLUTION one))

(define (edge B4)
  (TYPE not-important)             (define (mesh-problem M349)
  (BOUNDARY-CONDITIONS fixed)        (DESCRIPTION b7)
  (LOADINGS no-loaded)               (SOLUTION one))
  (NEIGHBOUR-ZX-R B35)
  (NEIGHBOUR-ZX-L B35))
```

Notice that the objects belonging to sort *edge* have as values of some features (i.e. neighbour, opposite, etc) objects that are also of sort *edge*.

Results of the application of INDIE and DISC over the Mesh dataset can be respectively found in section 4.1.7 of chapter 5 and in section 4.1.7 of chapter 6.

8. Soybean Dataset

This dataset has been obtained from the Irvine ML Repository. Objects of this dataset are descriptions of soybean plant having some disease. Concretely, there are 19 solution classes corresponding to the different diseases. Each example is described by 35 features.

There are two datasets: small soybean and large soybean. Small soybean dataset is a subset of the large Soybean. It contains 49 examples that are also contained in large soybean dataset. None of these 49 examples has unknown values and there are only four solution classes: diaporthe stem canker, charcoal rot, rhizoctonia root rot and phytophtora rot. Instead, large soybean has 307 examples that can belong to 19 solution classes. Some features of the examples can have unknown values and others can have the special value DNA (*does not appear*).

Commonly, examples in this domain are represented as attribute-value vectors. We have represented the examples as feature terms belonging to the sort *soybean-problem* having two features: `description` and `solution`. The `description` feature contains the description of a soybean plant (that is represented as an object of the sort *soybean*) and the feature `solution` contains the solution class to which the described plant belongs. Figure A.14 shows the description of an example.

As we have explained in section 4.2.3 of chapter 5 when a feature has unknown value it does not appear in the feature term. Concerning the DNA value, we have handled it as any other value.

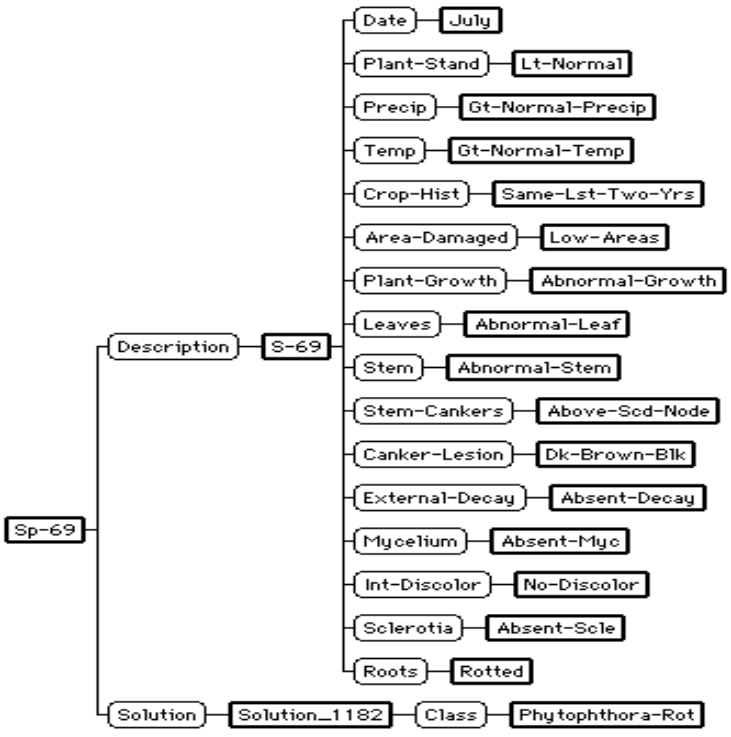


Figure A.14. Description of a soybean example.

9. Lymphography Dataset

This dataset has been obtained from the Irvine ML Repository. Objects of this dataset are descriptions of lymphographies. Each lymphography is described by 18 features and there is no features with unknown values. There are four solution classes: *malign-lymph*, *metastases*, *fibrosis* and *normal-find*. These solution classes have a very different number of examples, i.e. *malign-lymph* contains 61 examples, *metastases* contains 81 examples, *fibrosis* contains 4 examples and *normal-find* contains 2 examples.

Using feature terms we have defined each lymphography as a feature term belonging to the sort *lymphography-problem*. A *lymphography-problem* has two features (see figure A.15): `description` and `solution`. The value of the feature `description` is a feature term belonging to the sort *lymphography* that has 18 features. The value of the feature `solution` is the class to which the described lymphography belongs.

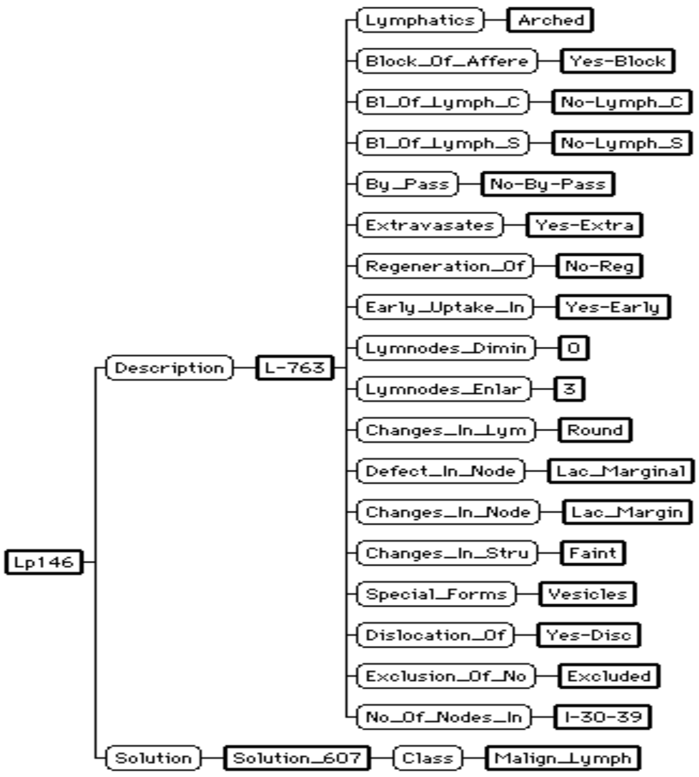


Figure A.15. Description of a Lymphography example.

APPENDIX B

Introduction to the Protein Purification domain

1. Introduction

Protein purification is the process that allows the isolation of one molecule among many others. Once the molecule of interest has been isolated, its structure, function, electrical and physical properties and behaviour can be analyzed. The development of techniques and methods for the separation and purification of biological macro-molecules (such as proteins) has been an important prerequisite for many of the advancements made in biosciences and biotechnology over the past three decades.

The purification process may be analytical or preparative. An analytical purification is made when there is a small amount of the molecule to purify. This kind of purification is used to adjust an appropriate purification process. A preparative purification process has as goal the purification of greater amount of a molecule.

A purification process is composed by one or several steps that can be included in three groups: fractionation and extraction, electrophoresis and chromatographic techniques.

The term chromatography groups a set of separation techniques based on the distribution of the molecules to be separated between two phases: one stationary phase and the other mobile. The chromatographic techniques allow the obtention of purified molecules using analytical or preparative processes.

The CHROMA application has a base of cases containing purification experiments over the biological macro-molecules called proteins[1] which have been purified using chromatographic techniques. In order to justify the utility of CHROMA, in the next sections we briefly explain several chromatographic techniques and then we discuss them.

2. Chromatographic Techniques

A chromatographic technique is based on the interaction of a stationary phase (chromatographic medium or the adsorbent) and a mobile phase that is a liquid or gas that moves through the stationary phase at a fixed flow rate. Molecules having a high tendency to stay in the stationary phase sill move through the system at a lower velocity than will those which favour the mobile phase. There are several configurations for the chromatography, such as the thin-layer chromatography (TLC) and the paper chromatography, the column chromatography probably the most common, and the high performance liquid chromatography (HPLC) that works at very high pressure conditions. In a column chromatography the stationary phase is packed into a tube through which the mobile phase (the eluent) is pumped (see figure B1). The various sample components move with different speeds through the column and are subsequently detected and collected at the end of the column. Column chromatography is very useful purifying molecules in biological extracts.

All the experiments in CHROMA have liquid mobile phase (might also be a gas) and the stationary phase is a porous matrix. This immobile solvent usually is a gel that constitutes the 90% of the stationary phase. In chromatography of proteins, the solvents are normally aqueous buffers and the gel-forming materials are usually composed by hydrophilic polymers. The shape, rigidity and particle size distribution profile of the gel matrix are important parameters which govern the performance of the stationary phase.

There is a wide variety of adsorbents allowing to exploit the different physico-chemical protein properties. Most important adsorbents are the *Gel Filtration, Ionic Exchange Chromatography*, *Chromatofocusing, Hydrophobic Interaction Chromatography, Reversed phase Chromatography, Covalent Chromatography, Immobilized Metal Ion Affinity Chromatography* and *Bioaffinity*.

[1] Proteins are polimeric structures based on the binding of single structural units, called aminoacids. All the proteins from all the species are a combination of the same 20 aminoacids. Proteins play key roles in almost all biological processes, this means, in life.

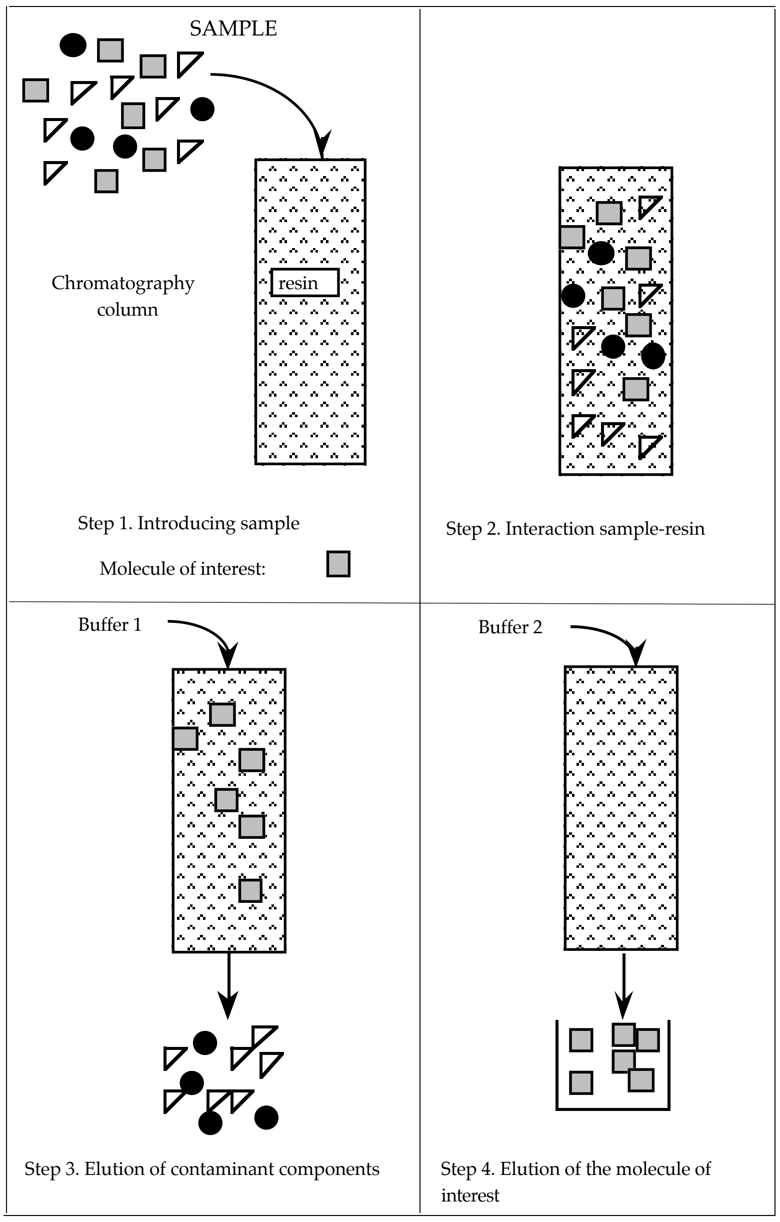

SAMPLE
Chromatography column
resin
Step 1. Introducing sample
Molecule of interest:
Step 2. Interaction sample-resin
Buffer 1
Buffer 2
Step 3. Elution of contaminant components
Step 4. Elution of the molecule of interest

Figure B.1. Column chromatography scheme. The sample containing the molecule of interest is introduced into one end of the column and it is collected at the other end. The different elution time allows the separation.

There are highly specific methods, such as those based on bioaffinity (e.g. antibody-antigen interaction) giving a highly pure protein in a single step. Normally, however, one has to combine several chromatographic methods to achieve complete purification of a protein from a crude biological extract[2].

In the follow we briefly describe the most used chromatographic techniques.

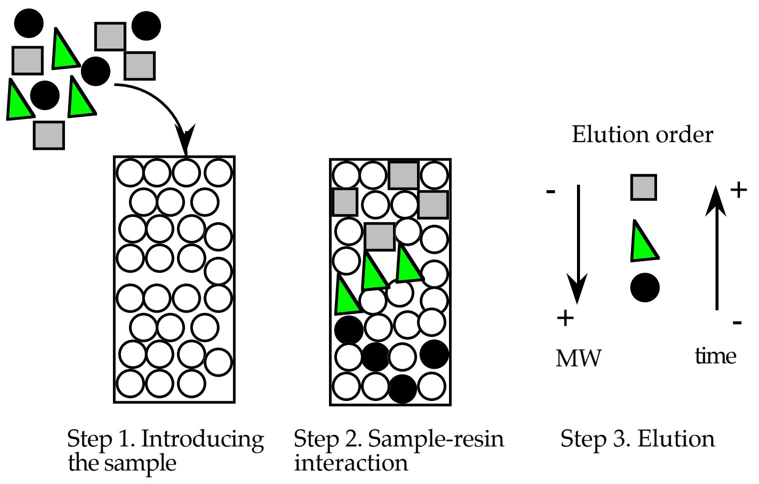

Elution order
-
+
MW
+
-
time
Step 1. Introducing the sample
Step 2. Sample-resin interaction
Step 3. Elution

Figure B.2. Scheme of a Gel Filtration Chromatography.

2.1. Gel Filtration Chromatography

Gel Filtration is a chromatographic technique that separates proteins according to its size (see figure B.2). Gel Filtration is an uncomplicated and straightforward technique, but there are some points worth consideration before starting the experimental work. The actual sample may require a special pH, solvent, additives or pre-treatment to yield a true solution. The next step is to select the gel that will cope with the chosen solvent and pH and that has a suitable separation range. Possible adsorption properties of the gel must also be considered. The nature of the separation and the sample may put demands on such parameters as resolution, separation time and sample load which in turn are partly dependent on the selected gel. These parameters are, however, also affected by the choice of column dimensions and the packing efficiency of the column. Obviously, for different separation problems, such as desalting, preparative purifications or analytical separations different requirements

[2] A crude extract is that directly obtained from the treatment (mashing, trituration) of an animal or vegetal tissue.

should be stressed. Economic factors and the possibilities of scaling up may also be important.

A Gel Filtration chromatography over lower size of proteins produces more efficient columns that can be used to obtain higher resolutions and/or faster separations. The Gel Filtration chromatography is useful to estimate the molecular weight of a protein, for desalting or fractionating solutions of proteins, and in analytical applications.

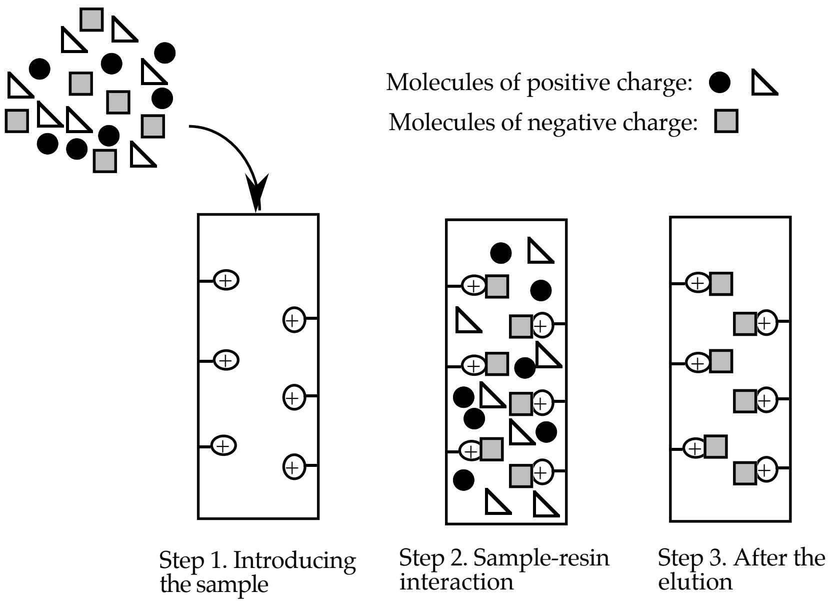

Molecules of positive charge: ● ◺
Molecules of negative charge: ▨
Step 1. Introducing the sample
Step 2. Sample-resin interaction
Step 3. After the elution

Figure B.3. Scheme of an Ion Exchange Chromatography.

2.2. Ion Exchange Chromatography

The basis for the ion exchange process (see figure B.3) is the competitive binding of ions of one kind, for example proteins, for ions of another kind, for example other proteins or salt ions of the same charge, to an oppositely charged chromatographic medium, the ion exchanger. The energy gained by the formation of an ionic bound between a protein and a charge on the stationary phase is expressed by the Coulomb's law. Therefore, if the two charges are of opposite sign there is a decrease in energy and if they are of the same sign there is an increase.

The interaction between the proteins and the ion exchanger depends on factors as the net charge and the surface charge distribution of the protein; the ionic strength and the nature of the particular ions in the solvent; the pH, and other additives to the solvent, such as organic solvents. The pH is one of the most important parameters which determine protein binding as it determines the effective charge on both the protein and the ion exchanger. The proteins are elued from the column in different order according to the pH. Thus, if using a pH the expected result is not obtained, the experiment may be repeated using a different pH. The control of the pH using a buffer salt is essential to reproduce the experiments. In fact, the results of the ion exchange process depends on the relative charges of the proteins in the sample at the pH in which the separation is made, since that the charge sign of a protein depends on the pH.

The binding of proteins to charged groups on the stationary phase competes with the binding of the ions in the solvent in a way that the proteins can be displaced from the ion exchanger due to the bind strengthens. There is no general rule to know the needed salt concentration to displace a protein with a certain net charge.

Altogether to the Coulomb's forces might be other interactions as the hydrogen bonds or hydrophobic interactions[3] producing additional effects for obtaining the separation of two similar proteins but these effects are difficult to predict and thus difficult to exploit them.

The ion exchange chromatography is specially indicated to purify great volumes of proteins due to its charge capacity and to its capability to concentrate proteins from the eluted solution.

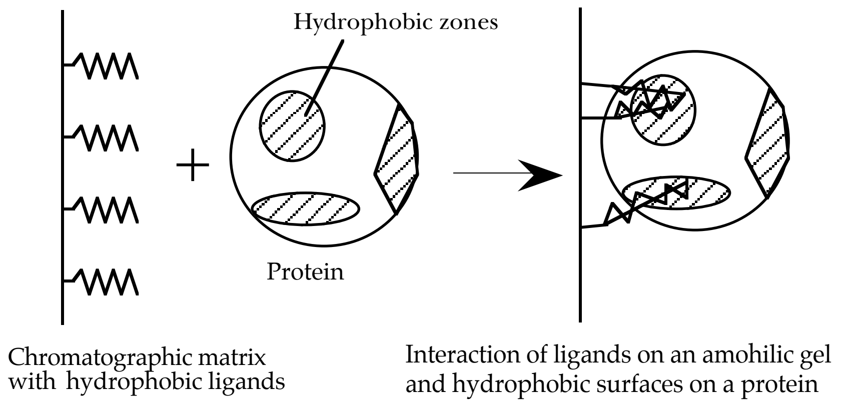

Hydrophobic zones
+
Protein
Chromatographic matrix with hydrophobic ligands
Interaction of ligands on an amohilic gel and hydrophobic surfaces on a protein

Figure B4. Scheme of a Hydrophobic Interaction Chromatography

[3] In a hydrogen bond one hydrogen atom is shared by other two atoms (usually oxigen and nitrogen). This kind of dynamical sharing, that some authors illustrate as a "menage a trois", is the basis of the bond.

2.3. Hydrophobic Interaction

Hydrophobic interactions are a phenomenon of great biological significance. They are one of the main forces that stabilize the three-dimensional structure of proteins. Hydrophobic interactions can be exploited and used as a means of separation. A definition of hydrophobicity is the repulsion between a non-polar compound and a polar environment such as water. Dissolving a non-polar substance in water is thermodynamically unfavourable due to the high surface tension. When a single hydrophobic compound is put into water, an energetically unfavourable conditions results. The hydrophobic compounds forces the surrounding water into an ordered structure as if it were forming a cavity. Because the solvent molecules close to the solute tend to gain order, the entropy decreases. Entropy is probably the driving force in hydrophobic interaction chromatography. An increase in the solvent entropy occurs when hydrophobic regions associate with an expulsion of water and reduction in the exposed surface available for solvation. High-ionic-strength mobile phases drive this association by increasing the surface tension of water, thus decreasing the amount of water molecules available to solvate the hydrophobic regions. This same phenomenon is the basis for the "salting out" of proteins by ammonium sulphate. The choice of salt for use in hydrophobic interaction chromatography is critical.

The hydrophobic interaction is of prime importance in biological systems, specially in folding globular proteins, the association of proteins subunits, the binding of many small molecules to proteins as in enzyme catalysis[4], regulation and transport across surfaces. It is also responsible for the self-association of phospholipids and other lipids to form the biological membrane bilayer and the binding of integral membrane proteins. The hydrophobic interaction is used for the binding of proteins to adsorbents with hydrophobic ligands. The degree of hydrophobicity of a protein is dependent on its amino acid sequences.

Hydrophobic Interaction Chromatography is a separation method based on the interaction between hydrophobic zones and protein molecules in a sample of a sample and an insoluble, immobilized hydrophobic group (those in the matrix, that is hydrophilic). The mobile phase is usually an aqueous salt solution. Separations on Hydrophobic Interaction Chromatography matrices are usually done in aqueous salt solutions, which generally are non denaturing conditions. Samples are loaded onto the matrix in a high-salt buffer and elution is by descending salt gradient. The elution can be made in three different ways: changing the salt concentration, changing the solvent polarity or adding solvents (organic

[4] Hydrophobic interactions are a weak and non-specific kind of bindig. Also known as Van der Waals interactions occurs when any type of atoms are as close as 3-4 Årmstrongs.

solvents are commonly used to alter the water polarity). Another way is to change the pH, but it is impossible to predict how this pH change can affect the strength of the interaction among the protein and the hydrophobic interaction gel.

The hydrophobic interaction chromatography is, in general, a mild method due to the stabilizing influence of salts. More labile proteins[5] upon contact with an hydrophobic interaction adsorbent, can change their structures. To do an hydrophobic interaction chromatography factors as the used salt concentration (when higher is the salt concentration produces a higher interaction), the additives changing the solvent polarity and the temperature have to be taken into account. The pH of the used buffers have a decisive influence in the adsorption of proteins to the gel. The only limitation for the pH value is the stability of both the protein to purify and the chromatographic matrix.

The hydrophobic interaction chromatography is based on a separation principle different of the used in other separation techniques. In that way, combining the hydrophobic interaction with other techniques can give a high purification degree. The high capacity of the hydrophobic interaction adsorbents makes them suitable for use at an early stage in a purification scheme.

2.4. Affinity

All the biological processes depend on specific interactions between molecules. These interactions might occur between a protein and low molecular weight substances, between bioinformative molecules (as hormones or transmitters) and receptors, and so on. Often biospecific interactions occur between two or several biopolymers, in particular proteins.

The affinity chromatography owes its name to the exploitation of these various biological affinities for adsorption to a solid phase. One of the members of the pair in the interaction, the *ligand* is immobilized on the solid phase, while the other, the *counterligand* (usually a protein) is adsorbed from the extract that is passing the chromatographic column.
It is essential that the ligand remains intact during the immobilization process. Also, the ligand has to be enough stable to carry out the projected affinity chromatography. Stability may be a problem when proteins are coupled at high pH, and the purity of the ligand should be as high as possible and in particular does not contain substances with functional groups that can react competitively in the immobilization.

[5] The stability of a protein in aqueous solution is extremely dependent on its own structure and composition. Some proteins are stable but others (labile proteins) lost its integrity in several hours.

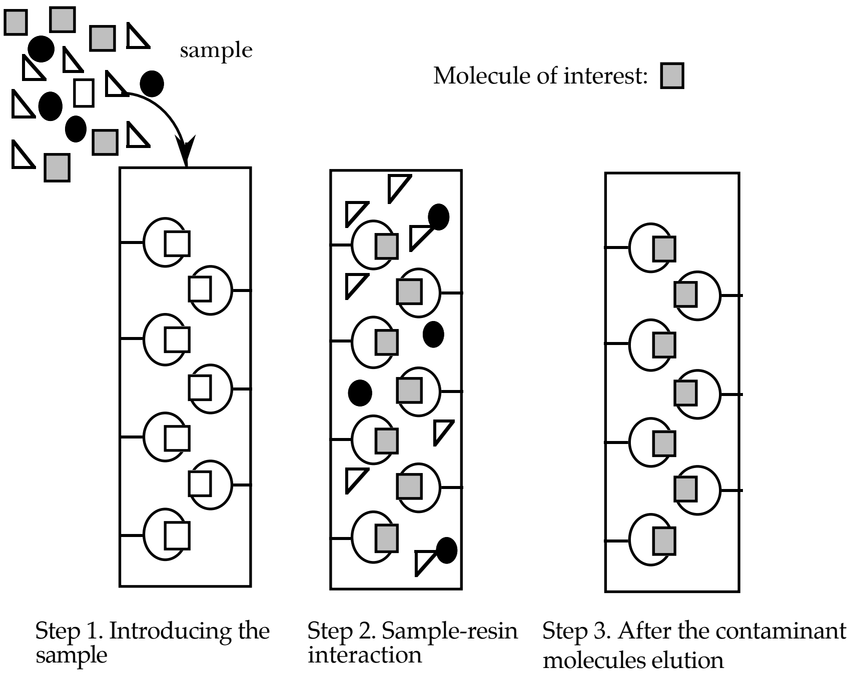

sample
Molecule of interest:
Step 1. Introducing the sample
Step 2. Sample-resin interaction
Step 3. After the contaminant molecules elution

Figure B5. Scheme of an Affinity chromatography.

In many cases, the affinity chromatography is a very powerful method, particularly when the protein of interest is a minor component (i.e. it has very small concentration) from a complex sample. The affinity chromatography depends on the functional properties, therefore the active and inactive forms may be separated. This chromatography is useful if the fractionement of the nucleic acids where the complementary base sequences may be used as ligands and, in the separation of the cell surface receptors are the basis of the affinity.

A field in which the affinity chromatography has been specially useful is acting over the antigen-antibody interactions, called *immunoadsorptions*. Very often this is the only way to purify a protein and it is specially attractive when there is a useful antibody at hand.

Main shortcoming of the affinity chromatography is that it often requires that the researcher synthesizes the adsorbent. Nevertheless, the methods to synthesise adsorbents are well known and easily adaptables. In preparing the affinity adsorbents we need to establish the binding between the matrix and the ligand as stable as possible. The stability will prevent the ligand leakage.

A property needing special consideration is the ligand-counterligand strength, since that if the strength is weak there is no adsorption and if the strength is high it is difficult to elute the adsorbed protein. It is important to search for conditions as the pH, salt concentration or inclusion of detergents or other substances promoving the complex dissociation without damage the active proteins. Frequently this is one of the main difficulties in the affinity chromatography.

A lot of affinity separations allows one step purifications. Nevertheless, sometimes is useful to include a preliminary step using a precipitation (if there is a lot of sample amount or a lot of contaminants) or a ion exchange chromatography to remove the big contaminants and consequently, to reduce the amount of processable material. In that way the resolution and the effects of the concentration of the affinity step are improved. When the sample is obtained from a tissue or from a cellular culture or it is a fermentation product, it is advisable to include some steps of solubilization, homogenization, extraction and/or centrifugation. To elute the affinity chromatography, changes of pH salinity or a specific ligand can be used.

APPENDIX C

Introduction to Marine Sponges Domain

1. Introduction

Species belonging to the animal kingdom can be distributed in two groups: Protozoa, including those simple organisms formed by only one cell; and Metazoa, including those organisms which body is formed by the union of several cells having a coordinated activity. Usually, metazoa are originated from one cell called *zygote*, which yields a pluricellular organism by means of successive subdivisions. The zygote origins a *morule*, having an empty spherical form that constitutes the *blastulae*. The blastulae is folded forming a bag of two layers called *gastrulae*. The internal gastrulae layer is called *endoderm* and the external layer is called *ectoderm*. This differentiation state basically characterises sponges and celenterea. All other metazoa follow their development state. During this development a third embrionary layer (called *mesoderm*) between the endoderm and the ectoderm appears.

It has been widely discussed the convenience of considering sponges as a separate animal subkingdom instead considering them as belonging to the metazoa group. Nevertheless, the analysis of the sponge development shows that this development is characteristic of the metazoa, although the cellular inversion observed in the blastulae is specific of the sponges. Another reason to consider the sponges as metazoa is the way in which they realise the *gametogenesis*[1]. Sponges present a wide variety of cellular kinds and, although they have not true organs, they present differentiated cellular sets showing morphological and functional well-determined properties. Thus sponges constitute the phylum Porifera into the metazoa.

1 gametogenesis is the production process of sexual cells.

Sponges are aquatic animals that live fixed to a substrate in both marine or fresh water habitats. The sponge body is constituted by a system of pores, ostia, canals and chambers called *aquiferus system*. The food is obtained from a water flow that is originated by an epithelium of flagellated cells called *choanocytes*. In the simplest case, the water pass through the sponge pores to the internal cavity, called *atrium*, and then goes out for a wide hole called oscule (see figure C.1). This process provides the sponge with food particles and oxygen. This way to obtain food is unique of the sponges. Choanocytes have a rounded body with an isolated flagellum and a collar of cytoplasmatic tentacles which are responsible for producing the water flow.

Most sponges have inorganic concretions of either calcium carbonate or silica called spicules, and also organic fibres can be observed. Both (organic and inorganic) elements in diverse combinations form the sponge skeleton.

The reproduction of sponges can be sexual (by means of larva production) or asexual (by producing *gemmules*). Gemmules produced in an asexual way are more resistant to adverse conditions of the environment.
The external aspect of the sponges is very varied, because there are a wide variety of shapes, colours, sizes and growing strategies. Usually, these characteristics are closely related to the external environment conditions. Into a group or species there may be high plasticity according to the environment conditions. Is for this reason that to separate the groups of the Porifera phylum, the skeleton morphology is used. In the next sections we will describe different aspects of the biology of sponges.

2. Morphology

In this section some general trends of the marine sponges, such as their structure, organisation levels and kinds of skeleton are described.

2.1. Structure and Level of Organisation

As all the metazoa, the sponge body is constituted by the external epithelium (ectoderm) and by the internal epithelium (endoderm). The ectoderm (see figure C.1) is formed by a layer of flat cells called *pinacocytes*, which form canals and allow the fixation of the sponge to the substrate. The endoderm is formed by flagellated cells called choanocytes. Between both epithelia there is the *mesohyl*[2] that can have different composition and amplitude but it always has skeletal material, cellular elements as the *scleroblasts* (that form the spicules) and the choanocytes that pump the water. The mesohyl also contains ameboide cells (*amebocytes*) transporting

[2] The mesohyl is composed by embrionary support tissue which cells have a big capacity of division by means of mitosis. The mesohyl is the main tissue from which the other support tissues of an organism are formed.

food particles and excretion products. Choanocytes are a kind of cell indicative and characteristic of the phylum.

An adult sponge can shows the following three organisation levels according to the water canalisation system (see figure C.1):

- *Asconoid*- This is the most simple type of sponges. The body is a simple tubular unit, with thin walls enclosing a central cavity which opens apically by a single osculum. The pinacoderm is interrupted by specialised pinacocytes which, in development, elongate and roll to enclose a cylindrical canal. These cells traverse the thin mesohyl and pierce the choanoderm between the bases of adjacent choanocites, thus placing the external medium in direct communication with the central choanocite layer.

- *Syconoid*- From an asconoid, folding of both pinacoderm and choanoderm produces a syconoid type. In a syconoid the inner choanoderm surface is amplified to line a series of projections which extend radially outward from the central cavity. The pinacoderm is folded outward to invest the projections. Choanocytes are now restricted to lining the choanocite chambers each of which opens to the central, atrial cavity by a wide aperture, the apopyle. The mesohyl has undergone relatively little thickening and thus inward flow of the water current can still be effected by way of porocytes, several of which open to each choanocyte chamber. The separation of inhalant surface and choanocyte chambers by the interposition of the cortex necessitates the development of an inhalant system. Most frequently this is a system of superficial ostia opening into inhalant lacunae, lined by pinacocytes, which then open by way of porocytes, or pores surrounded by several pinacocytes, prosopyles, to the choanocyte chambers. The atrial cavity is lined by a pinacoderm and opens by a single osculum.

- *Leuconoid*- Further folding of the choanoderm in the syconoid type is accompanied by subsivision of the flagellated surface into discrete spherical or oval chambers which are the typical flagellated, or choanocite chambers. Choanocyte chambers cluster in groups in the thickened mesohyl, and each chamber is serviced by two or more inhalant canals which open through propopyles and is drained by a single apopyle which leads to an excurrent canal. The excurrent canals coalesce to form larger channels and these converge toward exhalant apertures. This kind of organisation allows the development of very large sponges with high pumping efficiency.

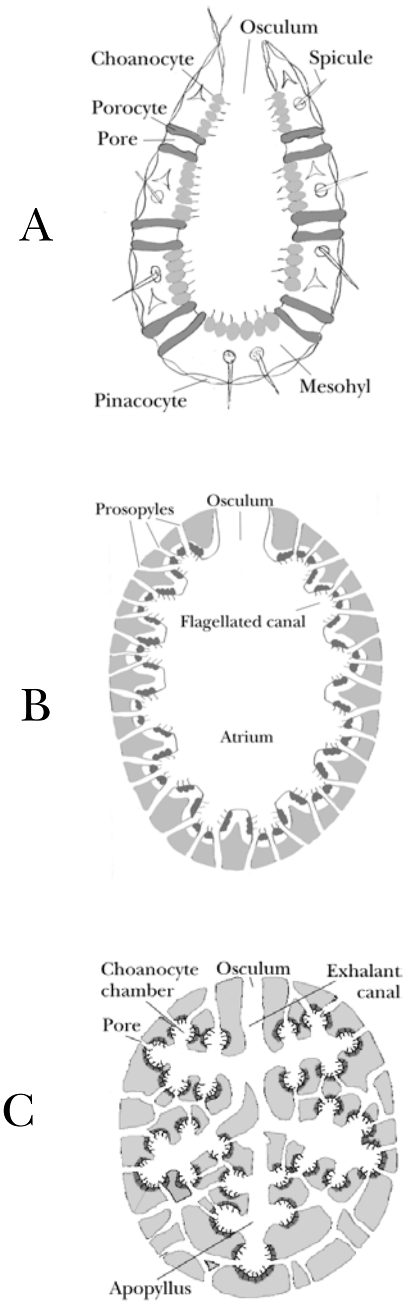


Figure C.1. Organisation levels of the sponges. A) Asconoid. B) Syconoid. C) Leuconoid.

Indepently on the organisation level of the canal system, all sponges need to maintain a continuos water flow from the exterior to the atrial cavity. A reduction in the pressure between the input and output holes produces a breath in effect over the water in the afferent conducts. This effect also transports food particles. When the water leaves the canal system still conserves the pressure excedent, which is profited to throw out the excretion products.

2.2. Sponge Skeleton

The main function of the skeleton is to maintain the body's structure. This function is achieved by the combination of inorganic spicules with some organic parts producing as result a solid texture. Another function attributed to the skeleton is to provide to the sponge a less attractive consistency for the predators.

Skeleton elements are very important to classify a sponge. An accurate analysis of the spicular geometry and its morphological characters are essential to determine the morphological and phylogenetic relations into the Porifera phylum. In the next sections several skeleton elements will be analysed.

2.2.1. Composition

The sponge skeleton has an internal and an external zone that can present a high diversity of combinations based on two kind of component: organic and inorganic. The organic part of the skeleton is mainly formed by *collagen*, the structural protein most widely distributed into the animal kingdom. In the sponge skeleton, there are two main types of collagenous organisation: scattered fibrilar collagen and spongin fibres. Thus, several dispositions can appear, for example fibres between 0 and 100 μm containing either silica spicules or foreign material (eg. sand, little debris, ...), fibres without spicules or thin fibres that are only visible at the ultrastructural level.

The spongin forms fibres less rigid than the pure collagen fibres, providing flexibility to the sponge body. Examples of sponges containing spongin are the typical bath sponges (*Euspongia officinalis* or *Hippospongia communis*) which skeleton is exclusively constituted by spongin.

The inorganic part of the sponge skeleton may be siliceous or calcareous. The siliceous skeleton is constituted by hydrated silica in a (no crystallised) amorph form. This siliceous skeleton can contain free spicules that either can be deposited in the mesohyl or can form nets together with spongin. The calcareous skeleton is composed of calcium carbonate mixed with magnesium carbonate. It is organised in several forms: as free spicules distributed along the mesohyl, forming nets of cimented spicules, or forming solid crystalline stores visible using a microscope.

2.2.2. The Spicules

Spicules are inorganic concretions of either calcium carbonate or silica. There are two kinds of spicules according to its function: the megascleres and the microscleres. Megascleres provide a solid structure supporting the skeleton, since they can take different dispositions as layers in different orientations. Instead, microscleres does not appear as a layer but as compactants among big spicules or reinforcing delicate surfaces as those of the internal canals.

The terminology used to name the spicules is related to its shape, size and ornamentation. Thus, the -*axon* termination is referred to the number of existing axis; -*actine* is referred to the radius finished in point; and the prefix *acantho*- is referred to other variations related to the apparition of spiny formations on the spicule surface.

According to this nomenclature, calcareous spicules could be classified in *diactines*, large and pointed; *triactines*, having three axis; and *tetractines* having four axis.

The silica sponges show a more wide variety of spicules. Among others, there are the *monaxons* (having a unique axis), the *tetraxons* (having four axis) and the *polyaxons* (having several axis starting from a common center. Monaxons and tetraxons are typical megascleres. Microscleres are more varied in form and they can characterise orders and families of Porifera. Three main classes of microscleres can be distinguished: *aster*, *sigma* and *raphid*. Aster is star-shaped due to a polyactine, and a divergent radius from a point or axis. There are several variants of the aster: *spiraster*, *spheraster*, *quiaster*, etc.

3. Nutrition of the Sponges

The nutrition process is carried out thanks to both the canal system and the continuos water flow. Sponge food is based on monocellular organisms, bacteria, organic detritus and soluble nutrients in coloidal status.

The global nutrition mechanism has three steps: ingestion, distribution and digestion of the food particles, and excretion. The ingestion of the particles can be made in different ways according to the size of the particle to be ingested. Nevertheless, it is not possible to establish a general form of operation for all the species. Thus, a way for ingestion is by the choanocytes collar. There are also described ingestion processes in cells belonging to the inhalant conducts and the *exopinacocytes*. Finally, the particles can directly pass to the mesohyl from an inhalant canal through a porocyte. Inside the sponge, the food particle is put in contact to mesohyl cells (the *archaeocytes*) which directly digest it.

The digestion is always intracellular. The digestion process has not been exhaustively studied but it seems clear that the archeocites are the main element in which the digestion is produced. The ingested particles flow to the mesohyl and then are stored in digestive vacuoles. When the vacuolae are full they are covered by an archeocite and incorporated to its interior where the digestion is made. On the other hand, the digestion may also be possible in the *phagosomes*[3] of the choanocytes. At present, there is still some uncertainty around the possible cellular sites of digestion and assimilation in choanocytes. Toxic or non digestible substances are evacuated by the exhalant canal system or by the sponge surface.

4. Reproduction

The sponges can present two forms of reproduction: sexual and axexual. In the next sections both forms are described. The lack of organs devoted to reproductive function makes it difficult the study of reproduction. Moreover, asynchrony in reproductive processes within individuals and within populations is common. thus, the knowledge on this area is still uncertain and is based on the observations made on different species trying to find the main lines for sponge reproduction.

4.1. Sexual reproduction

Sexual reproduction has been identified in all the studied species belonging to the Porifera phylum, even the concrete mechanism is different depending on the group. The gametogenesis process has been widely studied in several species but a complete scheme is not available. As consequence of the gametogenesis, the sponge can produce either sperm that are throw out at the water flow output or ovules which remain at the mesohyl. The release of the gametes (sperm or ovules) may be synchronous or asynchronous between individuals or between populations.

Many species are hermaphrodite, but usually produce oocytes or spermatocytes at different time. Gonads are not differentiated maintaining the general characteristics of the sponges that have not organs. Thus, all the functions are done at cellular level. The production of both male and female gametes isrelatedtothe choanocytes and archeocytes differentiation.

The spermatogenesis begins with the development of stem cells, called *spermatogonia*, from which the spermatic cysts are formed, and finally the spermatozoa maturation. The adult sponge has not reproductive tissue, thus, the sequence above begins with the differentiation of adult cells in the spermatogonia. In some species, the precursor cell is the choanocyte, but it is also possible that the precursor are cells in the mesohyl.

The oogenesis is also unclearly described. In the first studies choanocytes were considered as the precursor of the oocyte, but recent studies have suggested that the archaeocyte could undergo modifications during the oogenesis process. Specific studies in some species shown some

[3] Phagosomes are vacuoles formed at the time of phagocytosis.

mechanisms close to that above, although some processes still remains unclear. Thus, despite many factors related to the reproduction that have been studied, such as the temperature or size and age of the animal, there is a lack of knowledge in some essential points. Among others, why oogenesis and spermatogenesis are initiated, what causes the release of the gametes, or what is responsible of the reproductive period duration.

4.2. Asexual reproduction

The Porifera phylum has different kinds of asexual reproduction. The most common is based on the production of buds and regeneration processes, but in some species the production of larvae similar to those produced by sexual reproduction has been identified.

The production of buds has been observed in marine and fresh-water species. Currently, it is not clear which are the factors producing the asexual reproduction of sponges, nevertheless it seems that buds are a survival strategy in adverse environment conditions. Buds are initially formed by an agglomerate of archaeocytes with nutrient reserves and surrounded by spongin with characteristic spicules stores. This structure is able to survive on dry periods and low temperatures thanks to some vitelline platelets elaborated in the archaeocytes during the gemmule formation. Buds form a layer that remains in contact with the substract and consists in a mass of cells contained into a capsule. When other tissues are death, the buds are developed producing a little adult.

There is another kind of buds that produce larvae as those produced during the sexual reproduction. In fact, there are some sponge species capable to produce both sexual and asexual larvae.

Sponges are capable to regenerate from one or several fragments of an organism. The sponge reorganisation from an apparently random movement to form aggregate of cells has been observed in the study of cell suspensions in marine water. Then, these aggregates are surrounded by a cover of pinacocytes and is reorganised in its usual disposition.

5. Behaviour of the Sponges

Sponges have two states: larvary and adult. Almost all the sponges have a larvary state only known at the general level. The larvae measures from 50 μm to 5mm and it may be totally or partially ciliated. The larvae is very simple and it has not structures capable to select a concrete habitat. Nevertheless, the larvae state in different species present a high diversity of movements, longevity and reaction to contact and the light. These attributes indirectly represent a distribution form in respect to determined ambiental conditions in the adult state. In the larvary stady there are three factor influencing the movement and the fixation of the sponge larvae: light, gravity and turbulence.

The substrate that supports the sponge is a very documented trend of the sponges. In general, the larvae can be fixed in several surfaces, but most of species prefer hard natural substrates as rock, shells, coralin algae, etc.

As conclusion, we can deduce that the larvary state of the Porifera has different reactions to the external stimule. Thus, it seems obvious that the high habitat specialisation observed in adult poblations prove of a selective mortality of the larvae that have chosen indiscriminately a substrate. Nevertheless, it is important to remark that larvae of some species made a preselection of the substrate (Uriz, 1982).

Once the larvae has fixed and is converted into an adult individual it remains exposed to ambiental conditions that determine it their survival and expansion possibilities. Temperature seems one of the more significative factors in the larvae distribution. Therefore, the sponges are geographycally and bathymetrically distributed according to this factor. The sudden alterations of the temperature are mitigate with adaptations of the biological clicle allowing the survival: production of resistency forms. The body and spicular size into a species is bigger in lower temperatures. Some variations in the skeletal morphology correlated with the temperature are also observed.

The distribution of the sponges according to the depth is directly correlated to the light. The light quantity arriving to a determined depth also depends on other factors as the (organic or inorganic) material in suspension. Thus, indirectly varies according to the production and hydrodynamism of the water.

Sponges are distributed specially in the bathyal and in the circalitoral stages. The bathyal zone is the less deep of the aphytal zone in which the big poblations of the benthic community water have been disappeared. The circalitoral stage is the most depth zone where only calcified water are found. In the major irradiance zone can also be found some species usually associated with simbiont microalgues.

Porifera are mainly distributed in zones having a lower hydrodynamism with relative calm. Too violent waters cannot allow a correct fixation of the larvae and damage the adult body. Usually the sized is reduced and the form is constrained specially to the incrustant type. In too calm waters there are difficulties to maintain the continuos water flow which makes possible the physiology with filtrator character. It can exists an excessive sedimentation delays the normal development of the individual (usually by the obstruction of the ostia) and it is a limiting factor. In these conditions are possible several forms of the body (ramified, pinnate, flabelate or papillate).

The colonisation place of the Porifera is mainly the semidark part of the submerged caves. The population study has supported the bathymetric zonation according to the light factor, because they reproduce the open sea gradient but in the horizontal direction.

Sponges can live in very dark zones. When this occurs they became almost colorless. In these zones, sponges do not share the substrate with algae, thus it is usual to find rocky walls covered only by with sponges. Nevertheless, in extremely dark zones sponges disappear as well.

The trophyc and hydrodynamic conditions and the internal temperature of a cave have to be considered separately because they do not share the values of the exterior. In this case, it is not possible to generalise because each cave presents a community and a very particular characteristics conditioning the gradients of the physical factors mentioned above. Nevertheless it is not possible to state that the conditions of a semi-dark cave are a reflection of the depth environment having a hard substract. It is reasonable to think that larvae of species that are fixed in both geographical situations can have a similar evolution despite of the possible depth differences.

6. Poblational Dynamic

Dynamics of the Porifera populations is very varied depending in part on its own physiology, but also on problems derived of the coexistence with other organisms.

Each habitat presents specific difficulties for sedentary invertebrates. In principle, we can consider that the main objective of the larvae is to find an appropriate place to fix it on. Thus, the competition for the substrate can substantially influence the number, size and diversity of the species, and it constitutes the most limiting factor, even more than the alimentation sources.

To deal with this situation some species adopt the strategy of a quickly grow, achieving a good productivity despite a short life. These species are good colonisers. Other species have a longer life but a slow growing, and are placed in habitats that do not present problems of competition. Some species shows oscillations in the rhythm of productivity due to the fact that when they have a considerable size, it is more difficult for them to transport the food into the inner part.

7. Systematics

The Porifera phylum is composed by three main groups (classes): Hexactinellidae, Calcarea and Demospongiae. Hexactinellidae are the more primitive living sponges. Nowadays they live in deep waters but during the Jurassic and Cretacean period (195 and 65 millions of years ago) all the sponges lived on the littoral. Hexactinellidae have developed long spicules (of about 3 m) to use as anchorage. Hexactinellidae have a silica skeleton in which megascleres and microscleres can be clearly distinguished. Megascleres form a silica net in which there is disperse soft tissue. Typically, spicules have three main axis and, therefore, six pointed terminations: the hexactines.

Calcarea are small marine sponges that usually live in the littoral encrusted to stones. They have an skeleton constituted by calcium carbonate that is organised in spicules having two, three or four axis. These small sponges have a structure more complex that the Hexactinellidae, because the most of choanocytes are isolated in holes and they not cover the internal cavity.

Demospongiae are sponges from fresh or marine water. They represent the 95% of the actual sponge species. The Demospongiae are characterised by its inorganic skeleton formed by silica spicules (usually monoactines or tetractines) and an organic skeleton formed by collagen. The tetractines are considered as the primitive structure of the sponges belonging to this group. Nevertheless, some demospongiae have not skeleton, other have not megascleres, and other have not spicules.

References

Aamodt, A. (1991), A Knowledge-Intensive, Integrated Approach to Problem Solving and Sustained Learning. Ph. Dissertation. University of Trondheim. Norwegian Institute of Technology. Department of Electrical Engineering and Computer Science.

Aamodt, A. and Plaza, E. (1994), Case-based Reasoning: Foundational Issues, Methodological Variations and System Approaches. Artificial Intelligence Communications, (7)1. pp. 39-59.

Aarts, R. and Rousu, J. (1996), Towards CBR for Bioprocess Planning. In Advances in Case-based Reasoning. Third European Workshop on Case-based Reasoning. Lausanne, Switzerland. I. Smith and B. Faltings (eds). Lecture Notes in Artifical Intelligence. vol 1168. Springer Verlag. pp. 16-27.

Aha, D.W. (1991), Incremental Constructive Induction: An Instance-based Approach. Proceedings of the Eigth International Workshop on Machine Learning. pp. 117-121.

Aha, D.W., Kibler, D. and Albert M.K. (1991), Instance-based Learning Algorithms. Machine Learning, 6. pp. 37-66.

Aha, D.W. and Breslow, L.A. (1997), Refining Conversational Case Libraries. Proceedings of the Second ICCBR-97. Providence, RI, USA. July, 1997. Lecture Notes in Artificial Intelligence. D. Leake and E. Plaza (eds). CBR Research and Development. Springer Verlag.

Aït-Kaci, H. and Podelski, A. (1993), Towards a Meaning of LIFE. Journal Logic Programming, 16. pp. 195-234.

Akkermans, H., Van Harmelen, F., Schreiber, G. and Wielinga, B. (1993), A Formalisation of Knowledge-level Model for Knowledge Acquisition. International Journal of Intelligent Systems, 8. pp. 169-208.

Alterman, R. (1986), An Adaptive Planner. In Proceedings of AAAI-86. Cambridge, MA: AAAI Press/ MIT Press.

Alterman, R. (1988), Adaptive Planning. Cognitive Science 12. pp. 393-422.

Arcos, J.L. (1997), The NOOS representation Language. Ph. Dissertation. Universitat Politècnica de Catalunya.

Armengol, E. and Plaza, E. (1994), A knowledge level model of case-based reasoning. En Topics in case-based reasoning pp. 53-64. Lecture Notes in Artificial Intelligence, 837. S. Wess K.D Althoff and M. M. Richter (Eds).

Armengol, E. and Plaza, E. (1995), Explanation-based Learning: A knowledge level analysis. AI Review pp 19-35.

Armengol, E. and Plaza, E. (1997), Induction terms with INDIE. 9th European Conference on Machine Learning. Lecture Notes in Artificial Intelligence. Springer Verlag. pp. 33-48.

Bareiss, E.R. (1989), The Experimental Evaluation of a Case-based Learning Apprentice. In Proceedings from the Case-based Reasoning Workshop (DARPA), Pensacola Beach, Florida, May-June, 1989. Morgan Kauffmann. pp 162-167.

Bareiss, E.R., Porter, B.W. and Murray, K.S. (1989), Supporting start-to-finish Development of Knowledge Bases. Machine Learning, Vol. 4, 3/4. pp. 259-283.

Bartsch-Spörl, B. (1995), Towards the Integration of Case-based, Schema-based and Model-based Reasoning for Supporting Complex Design Tasks. In Case-based Reasoning Reasearch and Development. First International Conference, ICCBR-95, Sesimbra, Portugal. M. Veloso and A. Aamodt (eds). Lecture Notes in Artifical Intelligence. vol 1010. Springer Verlag. pp. 145-156.

Benjamins, R. and Pierret-Golbreich, C. (1996), Assumptions of Problem-Solving Methods. Lecture Notes in Artificial Intelligence, 1076. 9th European Knowledge Acquisition Workshop, EKAW-96. N. Shadbolt, K. O'Hara and G. Schreider (eds). pp. 1-16.

Bergadano, F. and Gunetti, D. (1993), An Interactive System to Learn Functional Logic Programs. Proceedings of the IJCAI. Cambery, France. pp. 1044-1049.

Bergadano, F. and Gunetti, D. (1996), Inductive Logic Programming, From Machine Learning to Software Engineering. Massachusetts Institute of Technology.

Bergadano, F., Giordana, A. and Saitta, L. (1988), Concept Acquisition in Noisy Enviroments. IEEE Transactions on Pattern Analysis and Machine Intelligence 10, pp. 555-578.

Bisson, G. (1992), Learning in FOL with a Similarity Measure. Proceedings of the AAAI-1992. pp. 82-87.

Börner, K. (1993). Structural similarity as guidance in case-based design. In: First European Workshop on Case-based Reasoning, Posters and Presentations, 1-5 November 1993. Vol. I. University of Kaiserslautern, pp. 14-19.

Bratko, I., Muggleton, S. and Varsek, A. (1991), Learning Qualitative Models of Dynamic Systems. Proceedings of the 8th International Workshop on Machine Learning. Morgan-Kaufmann, San Mateo, CA. pp. 385-388.

Breiman, L., Friedman, J.H., Olshen, R.A. and Stone, C.J. (1984), Classification and Regression Trees. Wadsworth, Belmont. CA

Bunke, H. and Messmer, B.T. (1994), Similarity Measures for Structured Representations. In Topics in Case-based Reasoning. EWCBR-94, pp. 106-118.

Buntine, W. (1988), Generalized Subsumption and Its Applications to Induction and Redundancy. Artificial Intelligence 36(2), pp. 149-176.

Cañamero, D. (1995), REPLACE: un modèle pour la reconaissance de plans. Ph. Dissertation. Université Paris-Sud. Centre d'Orsay.

Carbonell, J.G. (1986), Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In Machine Learnng: An Artificial Intelligence Approach. Volume II. R.S. Michalski, J.G. Carbonell and T.M. Mitchell eds. Tioga Publishing Company, Palo Alto, California. pp. 371-392.

Carbonell, J.G., Michalski, R.S. and Mitchell T.M. (1983), An overview of Machine Learning. In Machine Learnng: An Artificial Intelligence Approach. Volume I. R.S. Michalski, J.G. Carbonell and T.M. Mitchell eds. Tioga Publishing Company, Palo Alto, California. pp. 3-23

Carpenter, B. (1992), The Logic of Typed Feature Structures. Tracts in Theoretical Computer Science. Cambridge University Press. Cambridge, UK.

Cestnik, B., Kononenko, I. and Bratko, I. (1987), ASSISTANT 86: A Knowledge elicitation tool for sophisticated users. In Bratko and Lavrac Eds. Progress in Machine Learning. Wilmslow: Sigma Press. pp. 31-45.

Chan, P.K. and Stolfo, S.J. (1993), Meta-Learning for Multistrategy and Parallel Learning. Proceedings of the Second International Workshop on Machine learning. p. 150-165.

Chandrasekaran, B. (1986), Generic Tasks in Knowledge-based Reasoning: High-level Building Blocks for Expert System Design. IEEE Expert, 1. pp. 23-30.

Chandrasekaran, B. (1990), Design Problem Solving: A Task Analysis. AI-Magazine, Winter, pp. 59-71.

Clancey, W.J. (1985), Heuristic Classification. Artificial Intelligence, 27, pp. 289-350.

Clark, P. and Niblett, T. (1989), The CN2 Induction algorithm. Machine learning, 3, pp. 261-284

Cohen, W. (1994) Gramatically Biased Learning: Learning Logic Programs Using an Explicit Antecedent Description Language. Artificial Intelligence. pp. 301-366.

Conruyt, N., Manago, M., Le Renart, J. and Lévi, C. (1993), Une Méthode d'Acquisition de Conaissances pour la Classification et l'Identification d'Objects Biologiques. Application au Domain des Éponges Marines. Proccedings 13th International Symposium on Expert Systems and Applications, Avignon, France. Vol 1. pp. 485-495.

Cox, M.T. and Ram, A. (1994), Interacting Learning-Goals: Treating Learning as a Planning Task. Proceedings of the Second European Workshop on Case-based Reasoning. Chantilly, France. pp. 60-74.

Craw, S., Sleeman, D., Graner, N., Rissakis, M. and Sharma, S. (1992), CONSULTANT: Providing Advice for the Machine Learning Toolbox". Proceedings of the 1992 BCS Expert Systems Conference. M. Bramer (ed), Cambridge University Press.

Dasarathy, B.V. (1991), Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques. Los Alamitos, CA: IEEE Computer Society Press.

De Jong, G. and Mooney, R. (1986), Explanation-based Learning: An Alternative View. Machine Learning, 1. pp. 145-176.

De Raedt, L. (1992), Interactive Theory Revision: An Inductive Logic Programming Approach, Academic Press.

De Raedt, L. and Bruynooghe, M. (1993), A Theory of Clausal Discovery. Proceedings of the 13th International Join Conference on Artificial Intelligence, Morgan Kaufmann. pp. 1058-1063.

De Raedt, L., Lavrac, N. and Dzeroski, S. (1993), Multiple Predicate Learning. Proceedings of the Third International Workshop on Inductive

Logic Programming. p. 221-240. Bled, Slovenia. and also in Proceedings of the IJCAI-93 pp. 1037-1042. Cambery, France.

Dietterich, T. and Michalski, R. (1986), Learning to Predict Sequences. In Machine Learning: An Artificial Intelligence Approach. Volume II. R. Michalski, J. Carbonell and T. Mitchell eds. Morgan Kaufmann, Los Altos CA. p. 63-106.

Dolsak, B. and Muggleton, S. (1992), The Aplication of Inductive Logic Programming to Finite Element Mesh Design. In Inductive Logic programming. S. Muggleton (ed.). Academic Press, London. pp. 453-472.

Domingo, M. (1995), An Expert System Architecture for Identification in Biology. Monografies de l'IIIA. CSIC.

Emde, W. and Wettschereck, D. (1996), Relational Instance-based Learning. International Conference on Machine Learning. Lorenza Saitta (ed). Bari, Italy. pp. 122-130.

Flach, P. (1992), Logical Approaches to Machine Learning: An Overview. THINK, 1(2). pp. 25-36.

Friedman, J.H, Kohavi, R. and Yun, Y. (1996), Lazy Decision Trees, Proceedings of the AAAI-96. pp. 717-724.

Goel, A. (1991), A Model-based Approach to Case Adaptation. In Proceedings of the Eight International Machine Learning Workshop. San Mateo, CA: Morgan Kaufmann.

Goel, A. and Chandrasekaran, B. (1989), Use of Device Models in Adaptation of Design Cases. In Proceedings of the Second Workshop on Case-based Reasoning. K. J. Hammond (ed.). Pensacola Beach, Florida. Morgan Kaufmann.

Goodman, M. (1989), CBR in Battle Planning. In Proceedings of the Workshop on Case-based Reasoning (DARPA), Pensacola Beach, Florida. San Mateo, CA: Morgan Kaufmann.

Grobelnik, M. (1992), Markus: An Optimized Model Inference System. Proceedings of the ECAI Workshop on Logical Approaches to Machine Learning.

Haigh, K.Z. and Veloso, M. (1995), Route Planning by Analogy. In Case-based Reasoning Reasearch and Development. First International Conference, ICCBR-95, Sesimbra, Portugal. M. Veloso and A. Aamodt (eds). Lecture Notes in Artifical Intelligence. vol 1010. Springer Verlag. pp. 169-180.

Hammond, K.J. (1989), Case-based planning. Viewing planning as a memory task. Perspectives in Artificial Intelligence. Volume 1. Academic Press, Inc.

Hinrichs, T. and Kolodner, J. (1991), The Roles of Adaptation in Case-based Design. In Proceedings of AAAI-91. Cambridge, MA: AAAI Press/MIT Press. pp. 28-33.

Hinton, G.E. (1989), Connectionist Learning procedures. Artificial Intelligence, 40 (1-3). pp. 185-234.

Hurtley, N. (1995), Evaluating the Application of CBR in Mesh Design for Simulation Problems. In Case-based Reasoning Reasearch and Development. First International Conference, ICCBR-95, Sesimbra, Portugal. M. Veloso and A. Aamodt (eds). Lecture Notes in Artifical Intelligence. vol 1010. Springer Verlag. pp. 193-204.

Idestam-Almquist, P. (1993), Generalization under Implication using Or-introduction. In Proceedings of the 6th European Conference on Machine Learning. Vol 667, pp 56-64. Lecture Notes in Artificial Intelligence.

Janson, J.C. and Rydén, L. (eds). (1989). Protein Purification Principles, High Resolution Methods and Applications. VCH Publishers Inc.

Kibler, D. and Aha, D.W. (1987), Learning Representative Exemplars of Concepts: An Initial Case Study. Proceedings of the Fourth International Workshop on Machine Learning. University of California, Irvine. pp. 24-30

Kietz, J.U. and Wrobel, S. (1992), Controlling the Complexity of Learning in Logic Through Syntactic and Task-Oriented Models. Inductive Logic Programming, S Muggleton (ed.), Academic Press. pp. 335-359.

Kietz, J.U. and Morik, K. (1994), A Polynomial Approach to the Constructive Induction of Structural Knowledge. Machine Learning, 14, pp. 193-217.

Kirschenbaum, M, and Sterling, L. (1991), Refinement Strategies for Inductive Learning of Simple Prolog Programs. Proceedings of the 12th International Join Conference on Artificial Intelligence, Morgan-Kaufmann. Darling Harbour, Sidney, Australia. pp.757-761.

Kolodner, J. (1983), Maintaining Organization in a Dynamic Long-term Memory. Cognitive Science, Vol 7, pp. 243-280.

Kolodner, J.L. (1988), Proceedings of the Workshop on Case-based Reasoning (DARPA), Clearwater, FL. Morgan Kauffman, San Mateo, CA.

Koton, P. (1988) Reasoning about Evidence in Causal Explanation. In Proceedings of AAAI-88. Cambridge, MA: AAAI Press/MIT Press. pp. 256-261.

Kononenko, I. (1995), On Biases in Estimating Multi-Valued Attributes. Proceeding of the International Joint Conference on Artificial Intelligence. Montreal, Canada. pp. 1031-1040.

Lapointe, S. and Matwin, S. (1992), Sub-unification: A Tool for Efficient Induction of Recursive Programs. In Proceedings of the 9th International Workshop on Machine learning. Morgan Kaufmann. pp. 273-281.

Lavrac, N. and Dzeroski, S. (1994), Inductive Logic Programming Ellis Horwood Series in Artificil Intelligence. Ellis Horwood Limited.

Lavrac, N., Dzeroski, S. and Grobelnik, M. (1991), Learning Non-Recursive Definitions of Relations with LINUS. Proceedings of the 5th European Working Session on Learning, vol. 482 of Lecture Notes in Artificial Intelligence, Yves Kodratoff (ed.), Springer-Verlag. pp. 265-281.

Leake, D.B. (1992), Evaluating Explanations: A Content Theory. Northvale, NJ: Erlbaum.

Lebowitz, M. (1986), Integrated Learning: Controlling Explanation. Cognitive Science, 10. pp. 219-240.

Lenz, M., Burkhard, H.D. and Brückner, S. (1996), Applying Case Retrieval Nets to Diagnostic Tasks in Technical Domains. In Advances in Case-based Reasoning. Third European Workshop on Case-based Reasoning. Lausanne, Switzerland. I. Smith and B. Faltings (eds). Lecture Notes in Artifical Intelligence. vol 1168. Springer Verlag. pp. 219-233.

Ling, C. (1991), Inventing Necessary Theoretical Terms in Scientific Discovery and Inductive Logic Programming. Technical Report 302, Department of Computer Science, University of Western Ontario.

Lloyd, J. (1990), Computational Logic. Springer, Berlin. J. Lloyd ed.

López de Mántaras, R. (1991), A Distance-based Attribute Selection Measure for Decision Tree Induction. Machine Learning, 6, pp. 81-92.

Manago, M. (1989). Knowledge Intensive Induction, *Proceedings 6th International Machine Learning Workshop.* Morgan Kaufman.

Manago, M., Alhoff, K. D., Auriol, E., Traphöner, R., Wess, S., Conruyt, N., Maurer, F. (1993). Induction and reasoning from cases. *First European Workshop on Case-based reasoning.* Kaiserslautern: Germany..

Marcus, S. (editor) (1989), Machine Learning Journal, Special Issue on Knowledge Acquisition. Vol 4, n 3-4.

McDermott, J. (1988), Preliminary Steps Towards a Taxonomy of Problem Solving Methods. Automatic Knowledge Acquisition for Expert Systems. Marcus. S. (ed), Kluwer Academic Publishers.

Michalski, R.S. (1980), Pattern Recognition as rule-guided inductive inference. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2, pp. 349-361.

Michalski, R.S. (1983), A Theory and Methodology of Inductive Learning. In Machine Learning: An Artificial Intelligence Approach. Volume I. R.S. Michalski, J.G. Carbonell and T.M. Mitchell eds. Tioga Publishing Company, Palo Alto, California. pp. 83-134.

Michalski, R.S. (1991), Toward a Unified Theory of Learning: An Outline of Basic Ideas. Invited paper for the First World Conference on the Fundamentals of Artificial Intelligence. Paris, July.

Michalski, R. S., and Larson, J. B. (1978), Selection of Most Representative Training Examples and Incremental Generation of VL1 Hypotheses: The Underlying Methodology and the Description of Programs ESEL and AQ11. Report No. 867, Department of Computer Science, University of Illinois, Urbana.

Michalski, R.S., Mozetic, I., Hong, J. and Lavrac, N. (1986), The Multi-purpose Incremental Learning System AQ15 and its Testing Application on Three Medical Domains. In Proceedings of the Fifth National Conference on Artificial Intelligence, pp. 1041-1045. Morgan Kauffman, San Mateo, CA.

Michalski, R.S. and Tecuci, G. (eds), (1991), Proceedings of the First International workshop on Multistrategy Learning.

Mitchell, T.M. (1982) Generalization as Search. Artificial Intelligence, 18(2). pp. 203-226

Mitchell, T.M. (1991), The Need for Biases in Learning Generalizations. In Readings in Machine Learning. Morgan Kaufmann. pp. 184-191.

Mitchell, T.M., Keller, R.M. and Kedar-Cabelli, S.T. (1986), Explanation-based Generalization: A Unifying View. Machine Learning. Vol 1. pp. 47-80.

Mooney, R.J. and Ourston, D. (1991), A Multistrategy Approach to Theory Refinement. Proceedings of the 1st International Workshop on Multistrategy Learning. pp. 115-130.

Morik, K. (1991), Balanced Cooperative Modelling. Proceedings of the 1st International Workshop on Multistrategy Learning. George Mason University, Fairfax, VA. pp. 65-80.

Morik, K., Causse, K. and Boswell, R. (1991), A Common Knowledge Representation Integrating Learning Tools. Proceedings of the 1st International Workshop on Multistrategy Learning. George Mason University, Fairfax, VA. pp. 65-80.

Morik, K., Wrobel, S., Kietz, J.U. and Emde, W. (1993), Knowledge Acquisition and Machine Learning. Theory, Methods and Applications. Academic Press.

Muggleton, S. (1987), Duce, An Oracle Based Approach to Constructive Induction. Proceedings of the 10th International Join Conference on Artificial Intelligence, Morgan-Kaufmann. pp. 287-292.

Muggleton, S. and Buntine, W. (1988), Machine Invention of First-Order Predicates by Inverting Resolution. Proceedings of the 5th International Conference on Machine Learning. Morgan-Kaufmann. pp. 339-352.

Muggleton, S., Bain, M., Hayes-Michie, J. and Michie, D. (1989), An Experimental Comparison of Human and Machine Learning Formalisms. In Proceedings of the Sixth International Workshop on Machine Learning, pp. 113-118. Morgan Kaufmann, San Mateo, CA.

Muggleton, S. and Feng, C. (1990), Efficient Induction of Logic Programs. Proceedings of the 1st Conference on Algorithmic Learning Theory. Ohmsha, Tokyo.

Muggleton, S., King, R. and Sternberg, M. (1992), Protein Secondary Structure Prediction Using Logic-based Machine Learning. Protein Engineering 5(7). pp 647-657.

Muggleton, S. and De Raedt, L. (1994), Inductive Logic Programming: Theory and Methods. Journal of Logic Programming, 19, 20, pp. 629-679.

Muñoz-Avila, H. and Hüllen, J. (1996), Feature Weighting by Explaining Case-based Planning Episodes. In Advances in Case-based Reasoning. Third European Workshop on Case-based Reasoning. Lausanne, Switzerland. I. Smith and B. Faltings (eds). Lecture Notes in Artifical Intelligence. vol 1168. Springer Verlag. pp. 280-294.

Navichandra, D. (1988), Case-based Reasoning in CYCLOPS, A Design Problem solver. In Proceedings Workshop on Case-based Reasoning (DARPA), Clearwater, Florida. San Mateo, CA: Morgan Kaufmann. pp. 286-301

Nédellec, C., Rouveirol, C, Adé, H., Bergadano, F. and Tausend, B. (1996), Declarative Bias in Inductive Logic Programming. In Advances in Inductive Logic Programming, L. De Raedt Ed. IOS Press. pp. 82-103.

Netten, B.D. and Vingerhoeds, R.A. (1995), Large-scale Fault Diagnosis for On-board Train systems. In Case-based Reasoning Reasearch and Development. First International Conference, ICCBR-95, Sesimbra, Portugal. M. Veloso and A. Aamodt (eds). Lecture Notes in Artifical Intelligence. vol 1010. Springer Verlag. pp. 67-76.

Newell, A. (1982), The knowledge level. Artificial Intelligence 18, pp. 87-127.

Orsvärn, K. (1996), Knowledge Modelling with Libraries of Task Decomposition Methods. Ph. Dissertation. Swedish Institute of Computer Science.

Osterlund, B.R. (1993), Fuzziness is a Virtue for the Informed. International Laboratory News. April, 1993. p. 10.

Pagallo, G. and Haussler, D. (1990), Two Algorithms that Learn DNF by Discovery Relevant Features. Proceedings of the Sixth International Workshop on Machine Learning. Ithaca, NY. pp. 119-123.

Pazzani, M. and Kibler, D. (1992), The Utility of Knowledge in Inductive Learning. Machine Learning 9. pp. 57-94.

Pirnat, V., Kononenko, I., Janc, T. and Bratko, I. (1989), Medical Analysis of Automatically Induced Diagnostic Rules. In Proceedings of the Second European Conference on Artificial Intelligence in Medicine. pp. 24-36, Springer, Berlin.

Plaza, E. (1995), Cases as terms: A feature term approach to the structured representation of cases. Lecture Notes in Artficial Intelligence. Springer. num. 1010, pp. 265-276.

Plaza, E., López de Mántaras, R. and Armengol, E. (1996), On the Importance of Similitude: An Entropy-based Assessment. Lecture Notes in Artificial Intelligence Springer, num. 1168, pp. 324-338.

Plotkin, G. (1969), A Note on Inductive Generalization. In Meltzer, B. and Michie, D. (eds). Machine Intelligence, 5. pp. 153-163. Edinburgh University Press, Edinburgh.

Portinale, L. and Torasso, P. (1995), ADAPTER: An Integrated Diagnostic System Combining Case-based and Abductive Reasoning. In Case-based Reasoning Reasearch and Development. First International Conference, ICCBR-95, Sesimbra, Portugal. M. Veloso and A. Aamodt (eds). Lecture Notes in Artifical Intelligence. vol 1010. Springer Verlag. pp. 277-288.

Puerta, A.R., Egar, J.W., Tu, S.W. and Musen, M.A. (1992), A Multiple-Method Knowledge Acquisition Shell for the Automatic Generation of Knowledge Acquistion Tools. Knowledge Acquisition, 4. pp. 171-196.

Puyol, J. (1994), Modularization, Uncertainty, Reflective Control and Deduction by Specialization in MILORD II, a Language for Knowledge-based Systems. PhD Thesis, Universitat Autónoma de Barcelona. Abril, 1994.

Quinlan,J.R. (1986), Induction of Decision Trees. Machine Learning, 1, pp. 81-106.

Quinlan, J.R. (1990), Learning Logical Definitions from Relations. Machine Lerning, 5, pp.239-266.

Quinlan, J.R. (1993), C4.5: Programs for Machine Learning. Morgan Kaufman, San Mateo, California.

Ragavan, H. and Rendell, L. (1994), Lookahead Feature Construction for Learning Hard Concepts. Proceedings of the 10th International Conference on Machine Learning. University of Massachusetts, Amh rest. pp. 252-259.

Rouveirol, C. (1992), Extensions of Inversion of Resolution Applied to Theory Completion. Inductive Logic Programming, S. Muggleton (ed.), Academic Press, London. pp. 63-92.

Rouveirol, C. and Albert, P. (1994), Knowledge Level Modelling of Generate and Test Learning Systems". Proceedings of MLnet Workshop on Knowledge Level Models of Learning. Van de Velde (ed), Catania (Italy)

Russell, S. (1990). The Use of Knowledge in Analogy and Induction. Morgan Kaufmann.

Sammut, C. and Banerji, R.B. (1986), Learning Concepts by Asking Questions. In Machine Learning: An Artificial Intelligence Approach, Vol. 2, R. Michalski, J. Carbonell and T. Mitchell (eds.), Kaufmann, Los Altos, CA, pp. 167-192.

Schmidt-Schauss, M. (1988), Implication of Clauses is Undecidable. Theoretical Computer Science, 59. pp. 287-296.

Schank, R. (1982), Dynamic Memory: A Theory of Learning in computers and People. New York: Cambridge Univ. Press.

Schank, R. and Abelson, R. (1977), Scripts, Goals and Understanding. Northvale, NJ: Erlbaum.

Shapiro, E.Y. (1983), Algorithmic Program Debugging, MIT Press.

Simpson, R.L. (1985), A Computer Model of Case-based Reasoning in Problem Solving: An Investigation in the Domain of Dispute Mediation. Technical Report GIT-ICS-85/18, Georgia Institute of Technology.

Sleeman, D. and White, S. (1996), A Multistrategy Knowledge Refinement and Acquisition Toolbox: Revisited. Research Report AUCS/TR9605. Department of Computing Science. University of Aberdeen.

Smith, E.E. and Medin, D.L. (1981), Categories and Concepts. Cambridge, MA. Harvard University Press.

Smyth, B. and Keane, M.T. (1995), Experiments on Adaptation-guided Retrieval in Case-based Design. In Case-based Reasoning Reasearch and Development. First International Conference, ICCBR-95, Sesimbra, Portugal. M. Veloso and A. Aamodt (eds). Lecture Notes in Artifical Intelligence. vol 1010. Springer Verlag. pp. 313-324.

Stahl, I (1996), Predicate Invention in Inductive Logic Programming. In Advances in Inductive Logic Programming. L. De Raedt (ed). IOS Press. pp. 34-47.

Steels, L. (1990), Components of Expertise. AI-Magazine. Summer. p. 28-49.

Steels, L. (1993), The Componential Framework and its Role in Reusability. In Second Generation Expert Systems. J.M. David, J.P. Krivine, and R. Simons (eds.), Springer Verlag, Berlin Heidelberg, pp. 273-298.

Studer, R., Erikson, H., Gennari, J., Tu, S., Fensel, D. and Musen, M. (1996), Ontologies and the Configuration of Problem Solving Methods. Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop. SRDG Publications, University of Calgary. Alberta, Canada. pp. 11.1-11.20

Surma, J. and Braunschweig, B. (1996), REPRO: Supporting Flowsheet Design by Case-base Retrieval. In Advances in Case-based Reasoning. Third European Workshop on Case-based Reasoning. Lausanne, Switzerland. I. Smith and B. Faltings (eds). Lecture Notes in Artifical Intelligence. vol 1168. Springer Verlag. pp. 400-412.

Sycara, K. (1988), Using Case-based Reasoning for Plan Adaptation and Repair. In Proceedings of the Workshop on Case-based Reasoning (DARPA), Clearwater, Florida. San Mateo, CA: Morgan Kaufmann.

Tecuci, G.D. (1991), Learning as Understanding the External World. Proceedings of the 1st International Workshop on Multistrategy Learning. pp. 49-64.

Thonnat, M. and Gandelin, M.H. (1988), An Expert System for the Automatic Classification and Description of Zooplanktons from Monocular Images. Proceedings 9th International Conference on Pattern Recognition. Rome. pp. 114-118.

Uriz, M.J. (1982), Morfologia y comportamiento de la larva parenquímula scopalina y lophylopoda (Demospongia:Halichondria) y formación del ragon. Investigaciones Pesqueras, 46(2). pp. 313-322.

Utgoff, P.E. (1986) Machine Learning of Inductive Bias. Kluwer Academic Publishers.

Utgoff, P.E. and Mitchell T.M. (1982), Acquisition of Appropriate Bias for Inductive Concept Learning. In Proceedings of the National Conference on Artificial Intelligence, pp. 414-417. Morgan Kaufmann, Los Altos, CA.

Van de Velde, W. (1993), Issues in Knowledge Level Modelling. In Second Generation Expert Systems. J.M. David, J.P. Krivine, and R. Simons (eds.), Springer Verlag, Berlin Heidelberg, pp. 210-231.

Van Marcke, K. (1988), KRS: An Object oriented Representation Language. In Revue d'Intelligence Artificielle, Vol. 1, n. 4, Hermes. pp. 43-68.

Van Marcke, K. (1990), A Generic Tutoring Evironment. Proceedings of the ECAI. pp. 655-660.

Vapnik, V.N. and Chervonenkis, Y.A. (1981), Necessary and Sufficient Conditions for the Uniform Convergence of Means to their Expectations. Theory of Probability and its Applications, 26. pp. 532-553.

Wanwelkenhuysen, J. and Rademakers, P. (1990), Mapping a Knowledge Level Analysis onto a Computational Framework. Proceedings of the ECAI. pp. 661-666.

Wettschereck, D., Aha, D.W. and Mohri, T. (1997), A Review and Empirical Evaluation of Feature Weighting Methods for a Class of Lazy Learning Algorithms. AI-Review vol 11. n. 1-5. Special Issue on Lazy Learning pp 273-314.

Wielinga, B.J., Schreiber. A.T. and Breuker, J.A. (1992), KADS: A Modelling Approach to Knowledge Engineering. Knowledge Acquisition, 4, pp. 5-53.

Wielinga, B., Van de Velde, W. Schreiber, G. and Akkermans, H. (1993), Towards a Unification of Knowledge Modelling Approaches. In Second Generation Expert Systems, J.M. David, J.P. Krivine, and R. Simons (eds.), Springer Verlag, Berlin Heidelberg, pp. 229-335.

Winston, P.H. (1975), Learning Structural Descriptions from Examples. In The Psychology of Computer Vision. Winston, P.H. (ed). McGraw-Hill, New York.

Wnek, J. and Michalski, R.S. (1991), Hypothesis-driven Constructive Induction in AQ17: A Method and Experiments. Proceedings of the International Joint Conference on Artificial Intelligence. Sidney, Australia.

Wooley, J.B. and Stone, N.D. (1987), Application of Artificial Intelligence to Systematics: SYSTEX- a prototype Expert System for Species Identification. Sist. Zool., 36(3). pp. 248-267.

Wrobel, S. (1988), Automatic Representation Adjustement in an Observational Discovery System. Proceedings of the 3rd European Workshop Session on Learning, Pitman, London. pp. 253-262.

Wrobel, S. (1996), First Order Theory Refinement. In Advances in Inductive Logic Programming. L. De Raedt (ed). IOS Press. pp. 14-33.

Zdrahal, Z. and Motta, E. (1996), Case-based Problem solving Methods for Parametric Design Tasks. In Advances in Case-based Reasoning. Third European Workshop on Case-based Reasoning. Lausanne, Switzerland. I. Smith and B. Faltings (eds). Lecture Notes in Artifical Intelligence. vol 1168. Springer Verlag. pp. 473-486.

Zhou, X.J.M and Dillon, T.S. (1995), Theoretical and Practical Considerations of Uncertainty and Complexity in Automated Knowledge Acquisition. IEEE Transactions on Knowledge and Data Engineering, vol 7, 5. pp. 699-712.