The Aleph system made easy

By

João Paulo Duarte Conceição

THESIS SUBMITTED IN THE INTEGRATED MASTER OF ELECTRIC ENGINEERING AND COMPUTING

in the

FACULTY OF ENGINEERING

of the

UNIVERSITY OF PORTO

June 2008

Advisors

Rui Camacho
rcamacho@fe.up.pt
Faculty of Engineering, University of Porto, Portugal

Lubomír Popelínský
popel@fi.muni.cz
Faculty of Informatics, Masaryk University, Czech Republic

Abstract

This dissertation presents the background knowledge required to understand the concept of ILP[1] in general and the Aleph[2] System in particular. To comprehend ILP it's necessary understand two major concepts: Data Mining (process of discovering patterns in data) and Machine Learning (computer programs that improve with experience). There are many languages to develop ILP systems, and, the Aleph system uses Prolog, a programming language associated with artificial intelligence and computational linguistics. About the Aleph System it will be explained its settings, modes, determinations and types. This dissertation goal is the development of a graphical user interface for the Aleph System. With this interface it will be easy for non ILP reseachers to perform data analysis with Aleph. The user will be able to create files to be read by Aleph without knowing Prolog. The results given by the interface (using a Prolog compiler) are presented in a language very close to English. The created inputs and given results can be saved in different files. Final conclusions about the development of this interface are presented in the end of this document.

[1]Inductive Logic Programming
[2]A Learning Engine for Proposing Hypotheses

Acknowledgements

There are some persons who have contributed one way or another to make this dissertation possible. I would like to express here my gratitude to them:

- I wish to thank Professors Rui Camacho and Lubomír Popelínský, supervisors of this dissertation, for their advice, guidance and patience;
- Thanks to Knowledge Discovery Group for all the useful information that they provided to me and also for sharing with me their experience;
- Thanks to Nelson Costa for the valuable suggestions, support and for his help;
- Special thanks to my family, in particular, my father, my mother and my grandmother for giving me me all the support to develop the dissertation under the Erasmus Programme.
- And last, but not least, many thanks and a warm gratitude to my girlfriend Leihla, for her patience and constructive criticism, for giving me support and strength to make all this come true.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context and Motivation

Nowadays we are surrounded by a huge amount of information. That quantity has the tendency to continue to increase. The hard disks of the computers have more capacity to store information and with the prices decreasing, it leads to a exponential store of information. Ubiquitous electronics record our decisions, our choices in the supermarket, our financial habits, our comings and goings.[1]. In general, all our choices are stored in databases. Data mining is primarily used today by companies with a strong consumer focus: retail, financial, communication, and marketing organizations.[2] The amounts of information makes it impossible to be analyzed by human experts. Automatic data analysis is therefore required. This situation lead to the appearance of Data Mining [1]. In Data Mining, the information is stored electronically and the searches are autonomously done by a computer based in some patterns with the objective of solve a problem and simultaneously understand the content of the database. Data mining can be viewed as a result of the natural evolution of information technology [3]. In this context were developed techniques and algorithms of Artificial Intelligence that permit the computers to learn. The primary goal of these techniques is automatically extract information from the database using computational and statistical methods. Simultaneously adopt these decisions to the problem we want to solve in order to optimize his functional procedure and give the possibility to get conclusions based on founded patterns. It is possible that hidden among large piles of data are important relationships and correlations. Machine

[1]We use Data Mining even when referring to the whole process of Knowledge Discovery in Databases – KDD

learning methods can often be used to extract these relationships [4]. In the research area of Machine Learning and the development of logical programs is Inductive Logic Programming (ILP) [5]. There are many ILP systems, although to use most of them it is necessary to know-how ILP programming works [6]. The ILP system called "A Learning Engine for Proposing Hypotheses" (Aleph) is one of these systems and is where this project takes place.

1.2 Goals

The primary objective of this dissertation is the study, project, development and test of an interface for the Aleph ILP system. This interface should be usable by any non ILP researcher and it should be possible to construct models easily, so that anyone with basic knowledge about informatics can use this system. With the Aleph system we may construct several models and show them to the user for him to choose. After the models are created, it's possible to compile them and obtain some results. The results are presented in a language very close to English. This language was choosen, because it's universal and can be understood by a large portion of population in the world.

The partial goals are:

- Study and understand the concepts of Data Mining and ILP;
- Study the Aleph System;
- Specify, develop and test the Aleph Interface for the Aleph System;

Chapter 2

Introduction to Data Mining and Machine Learning

2.1 Data Mining

The amount of data in the world, seems to go on and on increasing, without end in sight. Personal computers make it easy to save things that we previously have trashed. Progress in digital data acquisition and storage technology has resulted in the growth of huge databases [7]. The hard disks are getting bigger and bigger and inexpensive making easy to postpone decisions about what to do with all this stuff, solving the problem by just buying another disk and keeping it all. But as the volume of data increases, inexorably, the proportion of it that people understand decreases, alarmingly. Automatic data analysis is therefore required and this situation lead to the appearance of Data Mining.

People have been seeking patterns in data since human life began. Hunters seek patterns in animal migration behavior, farmers seek patterns in crop growth, politicians seek patterns in voter opinion, and lovers seek patterns in their partners responses [1]. The major reason that data mining has attracted a great deal of attention in information industry in recent years is due to the wide availability of huge amounts of data and the imminent need for turning such data into useful information and knowledge. The information and knowledge gained can be used for applications ranging from business management, production control, and market analysis, to engineering design and science exploration.

Data Mining derives its name from the similarities between searching for valuable business information in a large database [11]. Its objective is to solve problems by analyzing data already present in databases and is defined as the process of discovering patterns in that data. The process must be automatic or (more usually) semiautomatic. The patterns discovered must be meaningful in that they lead to some advantage, usually an economic advantage. The data is invariably present in substantial quantities [1]. Exists two kinds of patterns: one can been as black box whose innards are effectively incomprehensible and the other is seen as a transparent box whose construction reveals the structure of the pattern. The kind of patterns that can be examined and be used to inform future decisions we call it *structural.* This kind of patterns help us to explain something about the data. All these knowledge discovery in databases is described in Figure 2.1, and consists of an iterative sequence of the following steps: [3]

- data cleaning (to remove noise or irrelevant data);
- data integration (where multiple data sources may be combined);
- data selection (where data relevant to the analysis task are retrieved from the database);
- data mining (an essential process where intelligent methods are applied in order to extract data patterns);
- pattern evaluation (to identify the truly interesting patterns representing knowledge based on some interestingness measures;
- knowledge presentation (where visualization and knowledge presentation techniques are used to present the mined knowledge to the user).

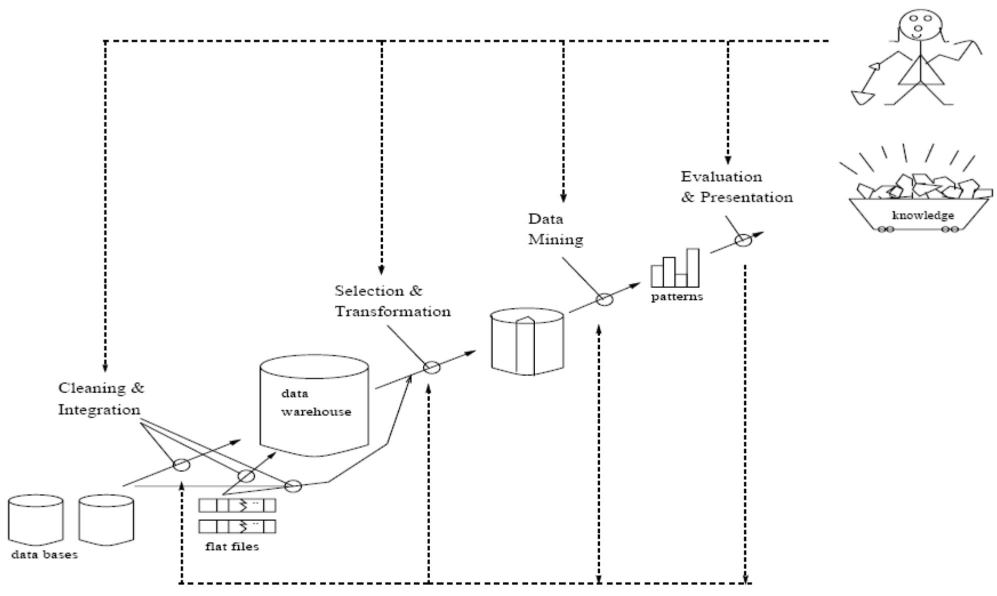


Figure 2.1: Data mining as a process of knowledge discovery [3].

After this process, the interesting patterns are presented to the user, and may be stored as new knowledge in the knowledge base. In this way, we can say that a data mining system is composed with six components, as the Figure 2.2 shows:

- database, data warehouse or other information repository (data cleaning and data integration will be applied to these kind of information repositories;
- database or data data warehouse server (responsible for fetching the relevant data, based on the data mining request);
- knowledge base (it's the domain knowledge that is used to evaluate the interestingness of resulting patterns);
- data mining engine (consists on a set of functional modules for tasks such as characterization, association analysis, classification, evolution and deviation analysis);
- pattern evaluation module (to identify the truly interesting patterns representing knowledge based on some interestingness measures);

- graphical user interface (this module it's used to do communication between the users and the data mining system in a interactive way. The user can specify a data mining query or task to help the search);

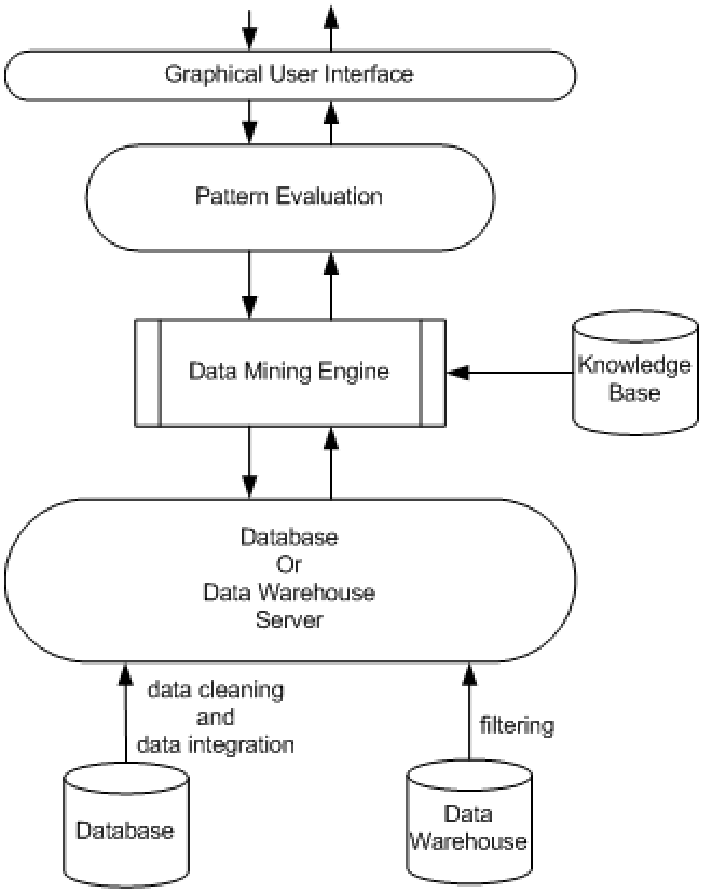


Figure 2.2: Typical data mining system architecture.

2.2 Machine Learning

The field of machine learning is concerned with the construction of computer programs that automatically improve with experience. Many successful applications of machine learning exist already, including systems that analyze past sales data to predict customer behavior, recognize faces or spoken speech, optimize robot behavior so that a task can be completed using minimum resources, and extract knowledge from bioinformatics data [12]. At the same time, there have been important advances in the theory and algorithms that form the foundations of this field [8].

Machine Learning draws on concepts and results from many fields, including statistics, artificial intelligence, philosophy, information theory, biology, cognitive science, computational complexity, and control theory. There are some important engineering reasons that make machine learning so important, some of them are:

- it is possible that in large piles of data are important hidden relationships and correlations. Machine learning methods can be used to extract these relationships;
- human designers often produce machines that do not work well as desired in the environments in which they are used. Machine learning methods can be used to improve the existing machine designs;
- machines can adapt to a changing environment reducing the need for constant redesign;
- the amount of knowledge available about certain tasks might be too large for explicit encoding by humans. Machines might be able to capture more knowledge than humans would want to write down;
- with the vocabulary changes, the discoveries made by humans and all the rest of new events in the world, redesigning the existing systems to new knowledge is not comfortable, but with machine learning methods it's possible to track much of that knowledge;

As mentioned before, machine learning usually refers to the changes in systems that perform tasks associated with artificial intelligence. The Figure 2.3 show the architecture of a typical artificial intelligence agent. This agent perceives and models its environment and computes appropriate actions, perhaps by anticipating their effects [4]. The changes made to any of the components show in the Figure 2.3 might count as learning. Depending on the system, different learning methods can be employed.

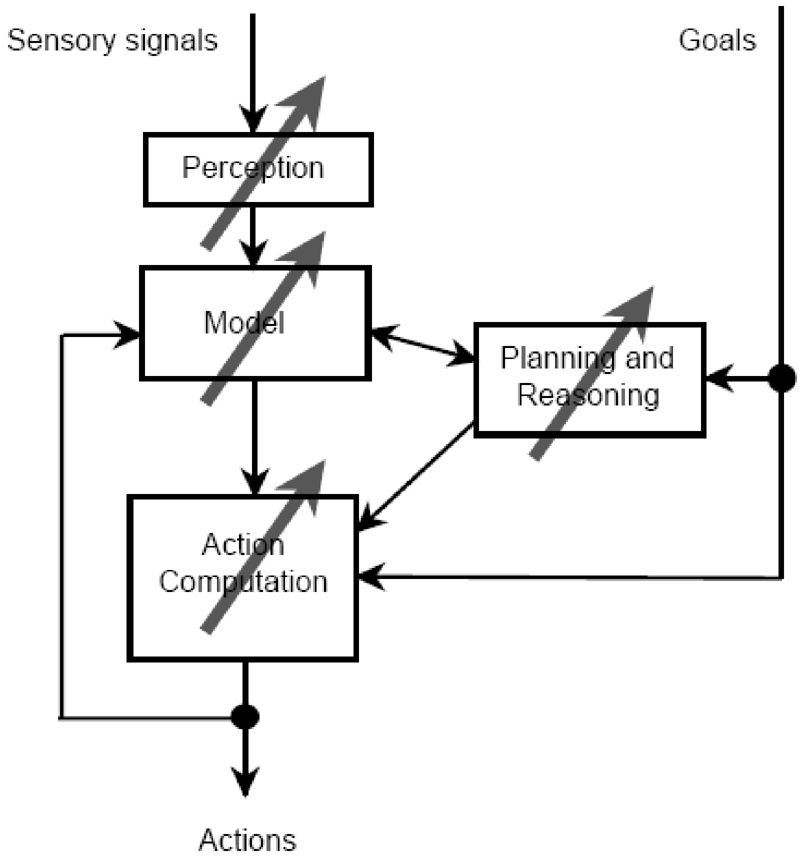


Figure 2.3: Artificial Intelligence System.[4]

2.3 Fielded Applications

This section is about data mining and machine learning used applications and will show you how important and useful is using it.

2.3.1 Screening Images

Scientists have been trying to detect oil slicks from satellite images to give early warning of ecological disasters and illegal dumping. To detect oil slicks in a manual way it's too expensive and require highly trained personnel. To solve this, a hazard detection system was developed in order to screen images for subsequent manual processing. Machine learning allows the system to be trained on examples of spills and non spills supplied by the user and lets him to control the trade off between undetected spills and false alarms. In this system the input will be a set of raw pixel images from a radar satellite and the output is a much smaller quantity of images with putative oil slicks marked by a colored color. The processing operations for this are:

- normalize the images with standard image processing operations;
- identification of suspicious dark regions;
- extraction of some attributes (size, shape, area, etc.) from each region;
- apply standard learning techniques to the resulting attributes.

2.3.2 Load Forecasting

It's important to determine future demand power as far in advance as possible in the electricity supply industry. In order to do this, an automated load forecasting has been developed and has been in use for the past decade to generate hourly forecasts for 2 days in advance. To create this automated and sophisticated load forecasting it was necessary to collect 15 years of data. Electric load shows periodicity at three frequencies:

- diurnal, where usage has an early morning minimum and midday and afternoon maxima;
- weekly, where demand is lower at weekends;
- seasonal, where increased demand during winter and summer for heating and cooling;

Special days such as Christmas, Thanks-giving and New Year's day have signification variation from the normal load and are each modeled separately by avering hourly loads for that day over the past 15 years. It was created a database with some fields, such as temperature, humidity , wind, speed and cloud for each hour of the 15 years, along with the difference between the actual load and that predicted by the model.
The conclusion taken is that the system as the same performance as trained human forecasters, but the system is much more faster, taking just some seconds to forecast a day, instead of hours by humans.

2.3.3 Diagnosis

Diagnosis is one of the principal application where machine learning is used to. Usually the handcrafted rules used in these kind of systems perform better that machine learning, but this last one can be useful in situations where the manual producing rules are too intensive to make. Some devices, such as motors and generators are inspected regularly by technicians to see if there's any problem with it. The primary goal of this model is not to see wheter or not a fault existe, but to diagnose the kind of fault. The attributes were run through an algorithm in order to produce a set of diagnostic rules. These rules were shown to an expert and he was not satisfied with it, because he couldn't relate it to his knowledge and experience, so it was necessary to had background knowledge after the generated rules. After this the results were very complex, although the expert liked them because now he could relate him with his knowledge.
After these tests the conclusions indicate us that the learned rules were superior to the handcrafted ones.

2.3.4 Other Applications

There are enumerous other applications of machine learning, in this section I'll briefly debrief about some of them.
Other machine application is when a costumer reports a problem and the company must decide what type of techician should be assigned for that ptoblem. In biomedicine machine learning is used to predict drug activity by analyzing the chemical properties of drugs and their three-dimensional structure. In chemistry it has been used to predict the structure of some organic compounds, using magnetic ressonance spectra. At last, but not least, machine learning is also being used to predict the human preferences on TV programs and also for intrusion detection, recognizing unusual patterns on some operations.

Chapter 3

Inductive Logic Programming

3.1 Introducing ILP

Inductive Logic Programming (ILP) is a research area at the intersection of Machine Learning and Logic Programming [5]. Its objective is to learn logical programs from examples and knowledge in the domain.

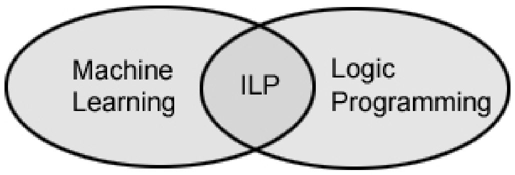


Figure 3.1: Intersection of Machine Learning and Logic Programming resulting ILP.

From Logic Programming ILP inherits its representation formalism, various techniques and a theoretical base. From Machine Learning inherits an experience and orientation approach for practical applications such as techniques and tools to induce hypotheses from examples and build intelligent learning machines [9]. These intelligent learning machines are learning programs which are able to change themselves in order to perform more efficiently and/or more accurately at a given task.

Using ILP you can get some advantages, such as [15]:

- get results in a quick way;

- ILP uses a powerful representing language;
- easy to understand the given results;
- it's possible to add information about the domain;
- there are a lot of domains where ILP can be used;
- almost all ILP systems are available online.

But at the same time ILP has some disadvantages, such as [15]:

- in complex situations it takes some time to get the results;
- it's necessary a experienced in ILP user to use the systems;
- the search space grows very quickly with the number of relations in the background knowledge.

The ILP differs from the Machine Learning methods because of the representation language and its ability to use knowledge in the domain. This knowledge has a very important place to the learner, which task is, to find from examples an unknown relationship in terms of relations already known from that domain.

The knowledge in the domain is used in the construction of hypotheses and it's a very important characteristic in ILP. In one hand, when this knowledge it's relevant it can substantially better the results of the learner in terms of precision, efficiency and its potential knowledge induced. On the other hand, some of this knowledge is irrelevant and will have opposite effects. The art of the ILP is to select and formulate the background knowledge to be used in the learner task [9]. ILP systems have been applied to various problem domains. Many applications benefit form the relational descriptions generated by the ILP systems [13].

3.2 Problems

In a general way ILP can be described by a knowledge theory from the initial background knowledge and some examples $E = E^+ \cup E^-$, where E^+

represents the positives examples and E^- represents the negatives examples of the concept to be learned.

The objective of the ILP is to induce an hypotheses H that with the given background knowledge B, explains the examples E^+ and is consistent with E^- [5]. Although in most of the problems, the given background knowledge, the examples and the hypotheses should satisfy a joint of syntactic restrictions S, called bias of the language. That bias language defines the space of formulas used to represent hypotheses and can be considered as part of the background knowledge. The empiric learner of a single concept (predicate) in ILP can be formulated in this way:

- A set of examples E described in a language Le with: positive examples, E^+ and negative examples, E^-;
- An unknown predicate p, specifying the relationship to be learned;
- A language description of hypotheses, Lh, specifying the syntactic restrictions in the definition of the predicate p;
- The bias S, that define the space of hypotheses;
- The theory of the background knowledge B, described in a language Lb, defining predicates that can be used in the definition of p and can give additional information about arguments from the examples of the target relation.
- A operator between Le and Lh with relationship to Lb which determine if an example it's covered by a closure expressed in Lh.

Find:

- A definition H for p, expressed in Lh, so that $B \wedge H = E^+$ and $B \wedge H \neq E^-$;

3.3 Language Bias

Any mechanism employed by a learning system to constrain the search for hypotheses is named bias [5]. The bias language defines the space of formulas used to represent hypotheses and can be considered as part of the background knowledge. Bias can either determine how the hypotheses space is searched (search bias) or determine the hypotheses space itself (language bias). [5]. In general there are three ways how to limit the size of the set generated by a refinement operator: to define bias, to accept assumptions on the quality of examples, or to use an oracle [14].

The search bias refers to how the system makes the search between the space of clauses. The exhaustive search is responsible to run all the space clauses and the heuristic searches indicates us which parts of the search should be used or ignored.

The bias language defines the space of formulas used to represent hypotheses. By selecting a stronger language bias the search space becomes smaller and learning more efficient; however, this may prevent the system from finding a solution which is not contained in the less expressive language [5]. The bias should be weak enough to allow complete and consistent programs and, simultaneously, enough to have a good performance.

There's also the validation bias which refers to when the system should stop. For instance, one condition to stop could be when a correct hypothese is found.

With the exception of the examples and the counterexamples of the concept, there are some factors that in one way or another influence the hypotheses selection that origin the bias. These factors are:

- the language which is used to describe the hypotheses;
- hypotheses space which the program can use;
- the procedure which define the order of how the hypotheses are considered;
- the condition that define if a search procedure should stop for a given hypothese or if should continue searching for a better one.

3.4 Completeness and Consistency of a Hypotheses

After we choose the examples and concepts, it's necessary to verify if a given example belongs to that concept. When that condition it's satisfied we say that the concept description covers the object description, or that the object description is covered by the concept description. [5]. The problem of the learner to a single concept C, described by examples, can be defined has:

Given a joint E, with positive and negative examples of a concept C, find a hypotheses H, described in a given language of description of concepts Lh, so that:

- All positive example $e \in E^+$ it's covered by H, and
- Neither negative example $e \in E^-$ it's covered by H.

To this test, a function $covers(H, e)$ can be defined. This function returns true if e it's covered by H, and false otherwise [9]. This function can redefined to a joint of examples in this way:

$$covers(H, E) = \{e \in E \mid covers(H, e) = true\}$$

A hypotheses H is complete with respect to examples E if it covers all the positive examples, i.e., $covers(H, E^+) = E^+$ [5]. A hypotheses H is consistent with respect to examples E if it covers none of the negative examples, i.e., if $covers(H, E^-) = 0$ [5]. There are four situations that can occur depending how the hypotheses H covers the negative and positive examples as shown in the Figure 3.2:

(a) H complete and consistent, cover all positive examples and neither negative examples;

(b) H incomplete and consistent, don't cover all positive examples and don't cover neither negative examples;

(c) H incomplete and consistent, cover all positive examples and cover some negative examples;

(d) H incomplete and inconsistent, don't cover all positive examples and cover some negative examples.

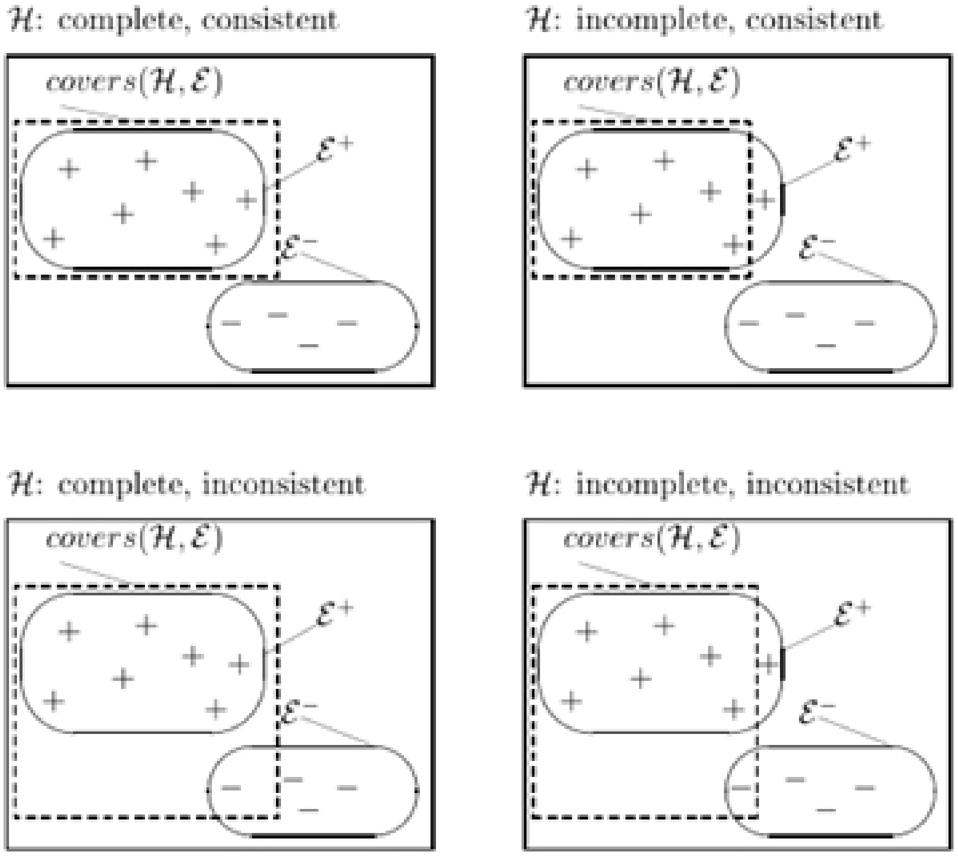

$\mathcal{H}$: complete, consistent
$covers(\mathcal{H},\mathcal{E})$
$\mathcal{E}^+$
$\mathcal{E}^-$
$\mathcal{H}$: incomplete, consistent
$covers(\mathcal{H},\mathcal{E})$
$\mathcal{E}^+$
$\mathcal{E}^-$
$\mathcal{H}$: complete, inconsistent
$covers(\mathcal{H},\mathcal{E})$
$\mathcal{E}^+$
$\mathcal{E}^-$
$\mathcal{H}$: incomplete, inconsistent
$covers(\mathcal{H},\mathcal{E})$
$\mathcal{E}^+$
$\mathcal{E}^-$

Figure 3.2: Completeness and consistency of a hypotheses. [5]

The cover function can be redefined to consider also the background knowledge B:

$$covers(B, H, E) = covers(H \cup B, E)$$

When we consider the background knowledge, the completeness and consistency also have to be redefined as shown below.

A hypotheses H it's complete, in relation to the background knowledge B and to the examples E, if all positive examples are covered:

$$covers(B, H, E^+) = E^+$$

A hypotheses H it's consistent in relation to the background knowledge B and to the examples, if neither negative example it's covered:

$$covers(B, H, E^-) = 0$$

3.5 Predictive and Descriptive ILP

Predictive ILP is one of the most common tasks in ILP and has the objective to define rules to the learner [9]. Typically that task confines E, H and B. The problem of the predictive ILP is defined by:

Given a background knowledge in the domain B, hypotheses H and a joint of examples E, one example $e \in E$ is covered by H if: $B \cup H = e$.

To extract an explanation in this kind of task it's necessary completeness and consistency of the hypotheses. To permit incomplete and inconsistent theories the problems should be extended to include learner rules for unperfected data, such as decision trees.

The primary goal of descriptive ILP is the learning of a theory clause [9]. Typically descriptive ILP restricts B to a joint of defined clauses and H to a joint of clauses and positive examples. To extract a rigorous explanation it is necessary that all clauses C in H be true in some predefined model. Relaxing the extraction of an explanation in a search clause and permitting theories that satisfies other acceptations like similarity and associability the descriptive ILP can be extended to incorporate associated learning rules.

3.6 Dimensions

The ILP systems can be divided, following some basic characteristics, in various dimensions [9]:

1. Can learn one single concept or multiple concepts (predicates). In case of learn one single concept, the observations are from that examples of

the concept. In case of learn multiple concepts, its objective is to learn that definitions, and make relations between them.

2. Can be necessary that all examples be given before the process of the learner takes place (batch learner or non-incremental) or can use special individual training, given one by one, during the process of learning (incremental learner);

3. Can need a specialist or the user during the process of learning to verify the valid generalizations or classify new examples. In this case, the system is called interactive, otherwise, non-interactive.

4. Can make up new predicates. Doing that, amplifies the usable background knowledge and can be useful in the task of learn a concept. Those systems are called systems with inductive construction.

5. Can accept an empty hypotheses (theory), learning a concept from the beginning or can accept an initial hypotheses that is reviewed during the learning process.

6. Can use background knowledge in an intentional or extensional way. In one hand the extensional theory is represented only by facts (without variables), on the other hand the intentional theory have facts or variables leading to a reduced description of the concept.

Although the recently ILP systems are divided in two extremes: in one extremity are situated the non-interactive systems with non-incremental learning. These ones learn a unique predicate from the beginning and are called empirical ILP systems. In the other extremity stay the interactive systems and theory reviewers that learn multiple predicates and are called interactive ILP systems. The empirical ILP systems tend to learn a single predicate using a large collection of examples [9]. The interactive ILP systems learn multiple predicates from a joint of examples and consults the user [9].

3.7 Description of ILP Systems

In this section will be described some ILP systems and their basic characteristics.

FOIL - it's an empirical system that learn multiple predicates from a non-interactive and non-incremental mode, realizing a top-down search in the hypotheses space.

Progol - an empirical system that can learn multiple predicates in a non-interactive and non-incremental way. Realize searches from general to specific from a top-down approach.

GOLEM - empirical system that learn one unique predicate at a time from a non-interactive and non-incremental way using bottom-up search.

FORS - empirical system that realizes prediction from examples and background knowledge in problems from classes with real values. It learns a unique predicate at a time from a non-interactive and non-incremental way doing a top-down search in the hypotheses space.

MIS - an interactive system and theory reviewer. It learns a definition of multiple predicates in a incremental way. Realize top-down search and it was the first ILP accepting background knowledge from an intentional and extensional way.

Tilde - it's a learning system based on decision trees. These trees can be used to classify new examples or transform in a logical program.

LINUS - an empirical, non-interactive and non-incremental ILP system. It transforms ILP systems to a attribute-value representation.

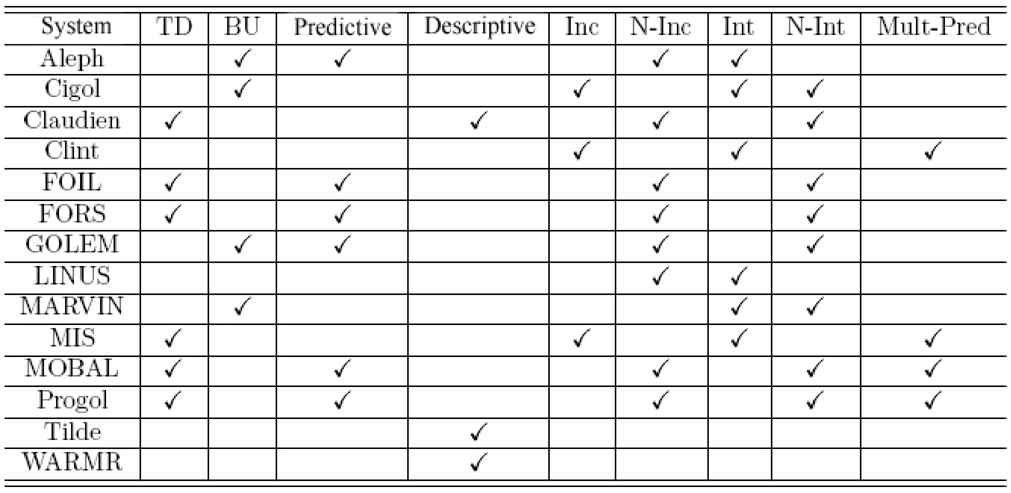
The Figure 3.3 shows the characteristics in the various ILP systems. This table is composed by the next columns:

1. System: name of the system;
2. TD: if the system uses a top-down search;
3. BU: if the system uses a bottom-up search;
4. Predictive: if the finding task of knowledge is predictive. In this case, classification rules can be generated;
5. Descriptive: if the finding task of knowledge is descriptive. In this case, only true properties from the examples are observed;

6. Inc and N-Inc: if the system uses incremental or non-incremental learning, respectively;

7. Int and N-Int: if the system is the type interactive or non-interactive, respectively;

8. Mult-Pred: if the system can learn multiple predicates.



System	TD	BU	Predictive	Descriptive	Inc	N-Inc	Int	N-Int	Mult-Pred
Aleph		✓	✓			✓	✓		
Cigol		✓			✓		✓	✓	
Claudien	✓			✓		✓		✓	
Clint					✓		✓		✓
FOIL	✓		✓			✓		✓	
FORS	✓		✓			✓		✓	
GOLEM		✓	✓			✓		✓	
LINUS						✓	✓		
MARVIN		✓					✓	✓	
MIS	✓				✓		✓		✓
MOBAL	✓		✓			✓		✓	✓
Progol	✓		✓			✓		✓	✓
Tilde				✓					
WARMR				✓					

Figure 3.3: Characteristics of various ILP systems.[9]

Chapter 4

A Learning Engine for Proposing Hypotheses

4.1 Aleph System

The A Learning Engine for Proposing Hypotheses (Aleph) System was developed to be a prototype to explore ideas in ILP and was written in P-Prolog. Since then, the implementation has evolved to emulate some of the functionality of several other ILP systems. Some these of relevance to Aleph are: CProgol, FOIL, FORS, Indlog, MIDOS, SRT, Tilde e WARMR [10]. The Aleph has a powerful representation language that allows to represent complex expressions and simultaneously incorporate new background knowledge easily. Aleph also let choose the order of generation of the rules, change the evaluation function and the search order [9]. Allied to all this characteristics the Aleph system is open source making it a powerful resource to all ILP researchers.

4.2 Basic Aleph Algorithm

The Aleph follows a very simple procedure that can be described in 4 steps [10]:

1. Select an initial example to be generalized. When there are not more examples, stop;

2. Construction of the more specific clause based on the restrictions language and the example selected in the last procedure. To this clause, we call bottom clause. To this step we call saturation.

3. Search for a more general clause than the bottom clause. These searches use the algorithm Branch-and-Bound. To this step, we call reduction.

4. Add the best clause to the theory, remove all redundant examples and return to step 1.

4.3 Requirements

The Aleph uses three files to construct a theory. In order to work properly, these three files should all have the same name. These are:

- file.b: contains the background knowledge (intentional and extensional), the search and language restrictions and restrictions in the types and parameters. All this content is in the form of Prolog clauses. This file can also contain any directives understood by the Prolog compiler being used;

- file.f: contains the positive examples (only facts without variables) to be learned with Aleph;

- file.n: contains the negative examples (only facts without variables). This file may not exist (Aleph can learn only by positive examples).

In order to use Aleph, a prolog compiler is needed. To compile Aleph it can be used one of these two platforms: Yap or SWI Prolog. Both of these compilers are open source and can be downloaded from the Internet[1].

4.4 Mode Declarations

The mode declarations stored in the file.b describe the relations (predicates) between the objects and the type of data. That declarations allows to inform Aleph if the relation can be used in the head (modeh declarations) or in the body (modeb declarations) of the generated rules [9]. The declaration modes also describe the kind of arguments for each predicate, and have the follow format:

[1]YAP: http://www.dcc.fc.up.pt/∼vsc/Yap/
SWI Prolog: http://www.swi-prolog.org/

mode(call_numbers, PredicateMode)

The call_numbers (also called recall), define the limit number of alternative instances for one predicate. A predicate instance it's a substitution of types for each variable or constant. The recall can be any positive number greater or equal to 1 or '*'. If it's known the limit of possible solutions for a particular instance, it's possible to define them by the recall. For instance, if we want to declare the predicate parent_of(P,D) the recall should be 2, because the daughter D, has a maximum of two parents P. In the same way, if the predicate was grandparents(GP,GD) the recall should be 4, because the granddaughter GD has a maximum of four grandparents GP. The recall '*' is used when there are no limits for the number of solutions to one instance.

The Modes indicates the predicate format, and can be described has:

predicate(ModeType1, ModeType2, , ModeTypen)

The ModeTypes can be organized in one of two ways: simple or structured. The simple modes can be one of:

- '+', specifying that when a predicate p appears in a clause, the corresponding argument it's an input variable;
- '-', specifying that the corresponding argument it's an output variable;
- '#', specifying that the corresponding argument it's a constant.

A structured ModeTypes is of the form f(...) where f is a function symbol, each argument of which is either a simple or structured ModeType [10]. An example of this kind of ModeType is:

:- mode(1,mem(+number,[+number|+list])).

Example: for the learning relation uncle_of(U,N) with the background knowledge parent_of(P,D) and sister_of(S1,S2), the mode declarations could be:

```
:- modeh(1,uncle_of(+person,+person))
:- modeb(*,parent_of(-person,+person))
:- modeb(*,parent_of(+person,-person))
:- modeb(*,sister_of(+person,-person))
```

The declaration modeh indicate the predicate that will compose the head of the rules [9]. For this case, modeh inform us that the head of the rules should be uncle_of(U,N) where U and N are from the type person. The symbol '+' that appears before the type indicates us that the argument of the predicate is a input variable. In this way, the head of the rules can be of the type uncle_of(U,N), and not, for instance, uncle_of(john,ana). The symbol '-' indicates us it's an output variable. Instead of '-', if the symbol '#' appeared, indicates us that the argument could be a constant. The modeb declaration indicate that the generated rules can have, in the body of the rules, the predicate parent_of(P,D), where P and D are from the type person. The first modeb declaration in the example can be used to add parent_of in the body of the rules and add one or more parent(s) to a daughter (observe that call_numbers have the value '*'). Similarly, the second declaration modeb let the predicate parent_of be used in the body of the rules to find one or more daughters of a parent. At last, the third modeb declaration can be used to find one or more sister of a person.

4.5 Types

Types have to be specified for every argument of all predicates to be used in constructing a hypotheses [10]. To the Aleph, these types are names and these names means facts. For example, the description of objects for the type person could be:

- person(john)
- person(leihla)
- person(richard)
- ...

Variables of different types are treated distinctly, even if one is a sub-type of the other [10].

4.6 Determinations

Determination statements declare the predicated that can be used to construct a hypotheses [10]. This declaration take the follow format:

determination(Target_Pred/Arity_t, Body_Pred/Arity_b)

The first argument is the name and arity of the target predicate [10]. It's the predicate that will appear in the head of the induced rule. The second argument it's the name of the predicate that can appear in the body of the rule. A possible determination for a relation called uncle_of(U,N) is:

determination(uncle_of/2, parent_of/2)

Typically, lots of declarations should be done for a target predicate. In case of non declared determinations, the Aleph doesn't construct any rule. Determinations are only allowed for 1 target predicate on any given run of Aleph: if multiple target determinations occur, the first one is chosen [10].

4.7 Positive and Negative Examples

The positive examples of the concept to be learned should be stored in the file with extension .f and the negative examples in the file with extension .n. For instance, to learn the concept uncle_of(U,N), we could have the follow positive examples in the file with extension .f:

- uncle_of(Sam,Henry)
- uncle_of(Martha,Henry)
- ...

And the follow negative examples in the file with extension .n:

- uncle_of(Lucy,Charles)
- uncle_of(Lucy,Dominic)
- ...

4.8 Parameters

The Aleph let us define a variety of restrictions to be learned in the hypotheses space, such as a search of new values and available parameters in that space [9]. The predicate set allows the user to define a value of the parameter Parameter:

set(Parameter,Value)

We can also get the current value of a parameter:

setting(Parameter,Value)

And for last, the predicate noset, change the current value for its pattern value.

noset(Parameter)

All the settings allowed for the Aleph System can be found in the Appendix A.

4.9 Other Characteristics

The Aleph has other important characteristics like:

- Instead of selecting one initial example to be generalized, it's possible to choose more than one. If we choose more than one initial example, it's created a bottom clause to each one of them. After the reduction step, the best of all reductions it's added to the theory;
- Let us construct the more specific clause, defining the place where the bottom clause it's constructed;
- The search clauses can be changed, using other strategies instead of using the Branch-and-Bound algorithm;
- It's possible to remove redundant examples to give a better perspective of the result clauses.

4.10 Using Aleph

Now that the information about the Aleph System was debriefed, it's important to know how to use it. In this document it will only be explain how to run it from the YAP compiler, although the usage of SWI Prolog it's very similar. To run and use Aleph, 7 steps are needed:

1. Download the file aleph.pl;
2. Download and install one of the two Prolog compilers (YAP or SWI Prolog);
3. Create the three files [2] .b, .f and .n;
4. Run the YAP Prolog compiler from a console terminal;
5. Now that the compiler is open we have to give him the file to compile. In this case the file to compile is aleph.pl:

   :- ['aleph.pl'].[3]

6. The next step is to load the files .b, .f and .n:

   read_all(filestem).

7. And for last the command induce will construct the theories:

   induce.

[2]These three files have to be all with the same name and the file .n is optional

[3]If the file aleph.pl it isn't in the same directory as the YAP executable, it's necessary to write the directory where the compiler can found this file. For instance, :- ['/home/aleph.pl'].

Chapter 5

Prolog

5.1 Overview

The name Prolog is an abbreviation for programmation en logique (French for programming in logic) and it was created in 1972. It is a logic programming language and is associated with artificial intelligence and computational linguistics. It was one of the first logic programming languages created and nowadays remains among in the most popular programming languages. Firstly this language had the purpose to work on language processing, but now it's used in many areas, such as: games, expert systems, automated answering systems and control systems. Prolog is very useful for working with databases, mathematics and language parsing applications.

Prolog is called a declarative language, i.e., the logic programming is expressed by relations and the execution is done by calling queries over these relations (defined by clauses). The called *term* is the Prolog single data type that allows to construct the relations and the queries. Logic programming is a programming paradigm based on mathematical logic. In this paradigm the programmer specifies relationships among data values (this constitutes a logic program) and then poses queries to the execution environment (usually an interactive interpreter) in order to see whether certain relationships hold [17].

The goal of the Prolog is to find a resolution refutation of one negated query. If this negated query is refuted successfully then the query is set to false. Prolog allows the use of impure predicates for checking if the value of a predicate may have some side effects, such as printing a value to the screen.

5.2 Syntax

The *term* is the Prolog single data type that allows to construct the relations and the queries. There are four kinds of terms in Prolog: atoms, numbers, variables and complex terms (or structures) [16].

An atom is composed by a sequence of characters that will be read by Prolog as a single unit. They are usually words written in Prolog code without any special syntax. Although if the atom has a space or use a capital letter then it has to be surrounded by single quotes in order to distinguish them from variables, for instance: 'the atom' or 'Atom'.

Numbers can be floats or integers. Real numbers aren't particularly important in typical Prolog applications [16]. Integers are useful for counting the elements of a list.

Variables are strings with letters, numbers and/or underscore characters. These kind of terms need to begin from a capital letter or with an underscore. The single underscore is called an *anonymous variable* and it means "any term". This type of variable does not represent the same value everywhere it occurs within a predicate definition.

The complex terms are composed by two arguments: an atom called "functor" and a number of arguments. The number of arguments is called the term's arity and an atom with arity of zero can be called an atom. The arguments are put in ordinary brackets, separated by commas, and placed after the functor [16]. The complex terms can be so complex as we want to, for instance, it's possible to have the structure:

walk(X,grandparent(grandparent(grandparent(pieter))))

In this case the functor is 'walk' and has two arguments: the variable 'X' and a complex term 'grandparent(grandparent(grandparent(pieter)))'. This structure has the functor 'grandparent' and another complex term 'grandparent(grandparent(pieter))' and so on.

5.3 Programming

A Prolog program is a set of procedures (the order is indifferent), each procedure consists of one or more clauses (the order of clauses is important) [18]. The objective of the Prolog programs is to describe relations using clauses. These clauses can be facts or rules. A rule consist in calls to predicates, which are called the rule's goals and they have the form:

```
Head :- Body.
```

This kind of rule can be read as "Head is true if Body is true". A fact is a rule without any body:

```
person(maria).
```

This fact is also equivalent to the rule:

```
person(maria) :- true.
```

After the facts and rules are built it's possible to make queries about those knowledge. For instance, making the query:

```
?- person(maria).
```

This query means "Is maria a person? " and the answer should be:

```
Yes
```

It's also possible to make a query such as:

```
?- person(X).
```

Which means “What things are persons? “ and the answer should be:

```
X = maria
```

It’s possible to add a program to the Prolog database using the *consult* command. The consult command adds the clauses and facts a the specified text file to the clauses and facts already stored in the Prolog database [18]. This command can be used in this way:

```
?- consult('name_of_the_file_with_the_program').
```

It’s also possible to reconsult a program file adding new procedures to the Prolog database. If there are procedures in the database with the same name as any procedure in the reconsulted file, then the existing one will be replaced. This command can be used like this:

```
?- reconsult('name_of_the_file_with_the_program').
```

The command *listing* give us the actual content of the Prolog database and it can be used in this way:

```
?- listing.
```

The Prolog program starts when one procedure of the loaded program is called. This procedure can be called in this way:

```
?- procedure_name(parameters).
```

Where *parameters* is the name of the procedure of the Prolog program. The *halt* command is used to stop the execution of the Prolog program:

```
?- halt.
```

5.4 Lists

In Prolog a list is represented between square brackets. An empty list is represented by []. When calling a predicate, we can create a list containing the elements a,b,c by typing:

```
[a,b,c].
```

It's also possible to append lists using the command *append* This command need three arguments where the two first arguments represent the two lists that we want to append and the third is the result of the append. For instance, if we write the query:

```
?- append([a,b,c],[d,e],X).
```

Writing this query, the answer should be:

```
X = [a,b,c,d,e]
```

It's also possible to define a list using [X|Y] and represents a list whose head is X, and whose tail (the rest of the list) is Y:

```
?- append([H|Tail],List,[H|NewTail]).
```

5.5 Working With Files

This section will show how to read and write from and to files. If we have a file with all predicates definitions in a format .pl it's possible to call it, using:

```
:- [allpredicates].
```

Where *allpredicates.pl* is the name of the file we want to read. If, for example, the predicate definitions provided by one of the files are already available, because it already was consulted once, Prolog still consults it again, overwriting the definitions in the database [16]. It's also possible to use the command *ensure_loaded* which will check if the file .pl has already been loaded. If not, the file is loaded, if yes, Prolog will check whether it has changed since last loading. If the file has been changed, he will read it, otherwise goes on processing the program. This command can be used has:

```
:- ensure_loaded([listpredicates]).
```

Besides reading files, it's also important to know how to write results to files. To write to a file, a stream is needed. Streams can be seen as connections to files. To open a file and connect it to a stream it can be done with:

```
?- open(+FileName,+Mode,-Stream).
```

The argument *FileName* is the of the file that the user wants to read. *Mode* it can be one of: read, write or append. In the first one, the file is opened for reading and the others are both for writing. Although in all these cases, the file is created in case it doesn't exist. After finishing the operations with the file, it should be closed by using the command *close*:

```
?- close(Stream).
```

Where *Stream* is the name of the stream associated to that file. In case, for instance, that we want to write something to a file the correct way is doing:

```
?- open(filetest, write, os), write(os, something_to_write), close(os).
```

Chapter 6

Aleph Graphical User Interface

6.1 Introduction and Requirements

The work project is the development of a graphical user interface for the Aleph System. This interface should allow the creation of new models and read existing ones as well. After reading or create new models, the interface should take some conclusions about these models using the YAP Prolog compiler. The primary objectives of this interface are:

- it should be easy for non ILP researchers to construct models;
- the user should be able to construct this models without knowing how Prolog works;
- a small explanation about the settings, modes and types of the Aleph should be presented to the user while he's working with it;
- after compile the models, the conclusions should be shown in a language very close to English.

It's possible to use this interface in Linux or Windows Operating Systems. There are some differences in the development of the interface for supporting both, specially when it refers to save, load files and the compile procedure as well. Although the usage for the user is the same.

To develop this interface, the Java programming language was used. It was chosen this language, because it's considered a much simpler and easy to use object-oriented programming language when compared to the popular programming language, C++. Partially modeled after C++, Java has replaced the complexity of multiple inheritance in C++ with a simple structure called interface, and also has eliminated the use of pointers. Allied to all these advantages, Java is one of the first programming languages to consider security as part of its design. Because of the use of Java, one of the requirements to the user is to have a Java Virtual Machine (JVM)[1] installed in the computer. A JVM is a set of computer programs and data structures which use a virtual machine to execute other computer programs and scripts.

6.2 Project Development Tool

The tool used for the development of the graphical user interface in Java was the Eclipse software. This tool is an open source[2] software and is an Integrated Development Environment (IDE) written primarily in Java. With this tool it's possible to install plug ins written for the Eclipse and it has allied some advantages, such as:

- it's possible to edit, compile, link and debug the source code files of the project;
- the project can have a large scale;
- no makefile is needed to run it;
- there are many plug ins for several purposes offered as freeware, shareware or commercial basis.

Eclipse has its basis on the Rich Client Platform (RCP), constituted by:

- a standard bundling framework called Equinox OSGi;
- Core platform responsible to run the plug ins;
- the Standard Widget Toolkit (SWT) which is a toolkit for use in designing applications with graphical user interfaces;

[1]It can be downloded from here: http://www.java.com/en/download/windows_ie.jsp?locale=en&host=www.java.com:80

[2]It can be downloded from here: http://www.eclipse.org/downloads/

- JFace which is a class viewer which provide some help for handling in common programming tasks;
- the Workbench providing views, editors, perspectives and wizards.

To use Eclipse software, a Java Runtime Environment (JRE)[3]. JRE is a group from Java Development Kit (JDK) containing the executables and important archives which constitute the platform Java. The JRE already includes the JVM.

6.3 Prolog Compiler

To compile the Aleph System, a Prolog compiler is required. There were two software options: using YAP or SWI-Prolog. For the development of this project, the YAP Prolog compiler was chosen to be attached on the interface. Both of the compilers are similar, although the SWI-Prolog uses a graphical interface and this is a reason to choose YAP instead the SWI-Prolog, because this way it's possible to use the compiler without showing it to the user.

YAP (Yet Another Prolog) has been developed since 1985 and it was written in assembly, C and Prolog. Nowadays the whole system is now written in C [19]. This compiler is compatible with ISO-Prolog standard, Quintus and SICStus Prolog and besides Aleph, YAP it's also used in another two applications: *FSA Utilities Toolbox*[4] and *SceX: A Symbolic Music Processing System*[5]. YAP is a Prolog compiler that works interactively, you can type a code and in a interactively way see its output when it is running. Allied to these features, YAP is open source[6], making it a powerful compiler for the Aleph System.

6.4 Adaptive Pattern

An Adaptive Pattern, or also called Wrapper is a type of software that is used to attach other software components. In simple words, a wrapper

[3]It can be downloaded from here: http://java.sun.com/j2se/1.4.2/download.html
[4]More information in: http://www.let.rug.nl/∼vannoord/Fsa/
[5]More information in http://www.ncc.up.pt/SceX/pnSceX_4.html
[6]It can be downloaded from here: http://www.dcc.fc.up.pt/∼vsc/Yap/downloads.html

encapsulates a single data source to make it usable in a more convenient way than the original unwrapped source. Wrappers can be used to present a simplified interface, to encapsulate diverse sources so that they all present a common interface, adding functionalities to the data source, or exposing some of its internal interfaces [20]. In other words, the adaptive pattern is useful in situations where an already existing class provides some or all services you need but does not use the interface you need.

All wrappers have the same basic logical model [20]:

- The application operate in a language X;
- The application get responses in model Y;
- The wrapped data source is operated in a language Z;
- The wrapped data source responds with results expressed in model W.

The objective of the wrapper is to convert the language X commands to language Z and the model W results to model Y. The differences between wrappers are on the models they support and in the sophistication of the functionality they provide.

A good example to better understand what is a wrapper is the use of a socket wrenche. A socket wrench is a tool that uses separate, removable sockets to fit many different sizes. In this case, a socket attaches to a ratchet, providing that the size of the drive is the same. Typical drive sizes in the United States are 1/2 and 1/4. Obviously, a 1/2 drive ratchet will not fit into a 1/4 drive socket unless an adapter is used. A 1/2 to 1/4 adapter has a 1/2 female connection to fit on the 1/2 drive ratchet, and a 1/4 male connection to fit in the 1/4 drive socket [21]. This can be seen in the Figure 6.1.

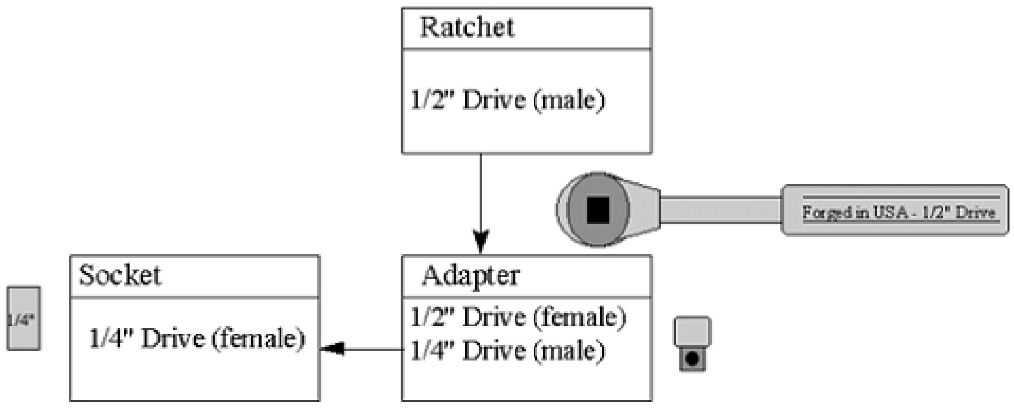


Figure 6.1: Adaptive Pattern Example.[21]

In the development of the Aleph graphical user interface, an adaptive pattern is needed in order to establish a link between the interface and the YAP compiler. The interface will use the YAP compiler in order to the interface show the results in a language very close to English. The solution to solve this problem in this project can't be called exactly an adaptive pattern, but kind of it. As mentioned before, YAP can receive some arguments when it's called and besides others, these arguments can be: the .pl file that you want to read, the files .b, .f and .n, the command to make the theories and other to write these theories to an output file. All this can be done in just one line when the YAP Prolog compiler is called:

```
yap -l aleph.pl -g read_all(filestem). -z induce. > out.txt
```

The argument -l compile the Prolog file, in this case *aleph.pl*, before entering the top-level. The argument -g run the goal, in this case *read_all*, where this goal will be converted from an atom to a Prolog term. The filestem is the name of the files .b, .f and .n. The argument -z means to run the goal, in this case *induce*, as top-level, where this goal will be converted from an atom to a Prolog term. At last, the symbol > indicate to write in a output file, in this case *out.txt*.

It's also important to mention that to this command work properly, the file aleph.pl has to be in the same directory as the YAP executable file. If not, it's possible to write the directory before the name of the file. The filestem (files .b, .n and .f) has to be in the same directory as the file aleph.pl. And for last it's also possible to define the directory for the output file before it's name.

After know how to write this command it was necessary to re-arrange a way to write this command in a console. The solution found to this problem was creating a file format .sh for Linux and a file .bat for Windows where their content is that command. The content of these two files is very similar but has some differences. The file yap.bat has the next content:

```
SET YAP_ILP_ROOT=%1\Yap-5.1.1\bin
%YAP_ILP_ROOT%\yap -l %YAP_ILP_ROOT%\aleph.pl -g read_all('%2').
-z induce. > %1\out.txt
%*
```

Where *%1* is the first argument received, which will be the directory where the Aleph interface was run and *%2* is the second argument, which will be the name of the filestem to compile.

In a similar way, the content of the file.sh is:

```
export YAP_ILP_ROOT=$1/Yap-5.1.2/ARCH
$YAP_ILP_ROOT/./startup -l $YAP_ILP_ROOT/aleph.pl -g 'read_all('$2').'
-z 'induce.' > $1/out.txt
$*
```

Having the files .sh and .bat created, it's possible to call them from a Java class:

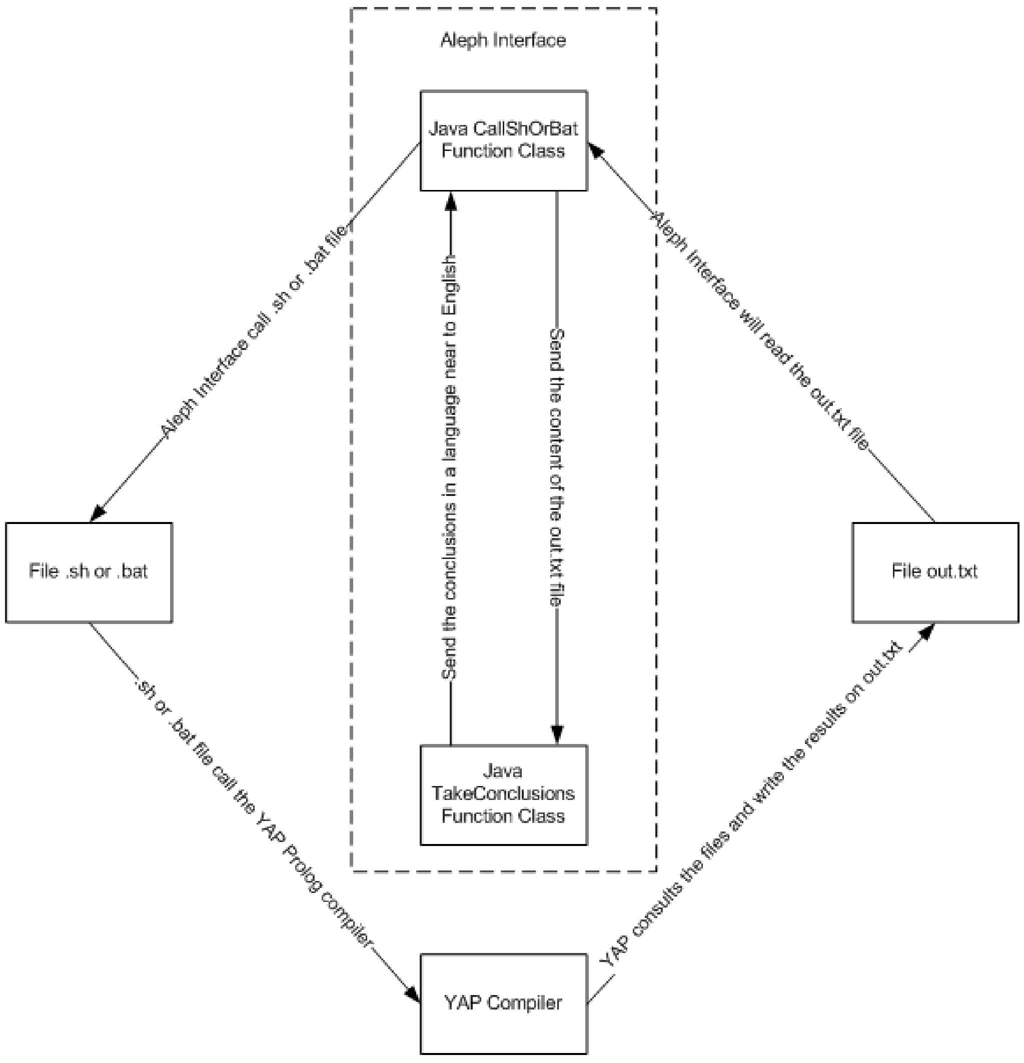
```
Runtime rt = Runtime.getRuntime();
rt.exec("yap.sh " + directory() + " " + selectedItem,null,new File(directory()));
```

These two Java lines run the file yap.sh located in the directory returned by the function *directory()*. This function returns the directory where the Aleph graphical user interface is being called. The variable *selectedItem* has the name of filestem. For instance, if the directory where the Aleph interface was run is: */home/workspace* and the filestem for compile (selectedItem) is *train*, the command sent to write in the console is:

```
/home/workspace/yap.sh /home/workspace/ train
```

Where */home/workspace/* is the first argument to enter in the file .sh and *train* is the second one.

After the file .sh has run, the interface will read the file out.txt, compile it inside the Aleph interface in order to write the conclusions in a language near English. The Figure 6.2 shows how all these procedures work:



Aleph Interface
Java CallShOrBat Function Class
Aleph Interface call .sh or .bat file
Send the conclusions in a language near to English
Send the content of the out.txt file
Aleph interface will read the out.txt file
File .sh or .bat
File out.txt
.sh or .bat file call the YAP Prolog compiler
Java TakeConclusions Function Class
YAP consults the files and write the results on out.txt
YAP Compiler

Figure 6.2: Adaptive Pattern for YAP.

6.5 Architecture

This section shows how the Aleph Interface Java code is organized. The code it is divided in two packages: gui and compiler. The first one is responsible for the creation and management of all the graphical user interface. The second one reads the created input files and using the YAP Prolog compiler shows the results to the user. These packages trade information and each one have classes which trade information as well. The Figure 6.3 shows how the code is organized and how the information is traded from each class and package:

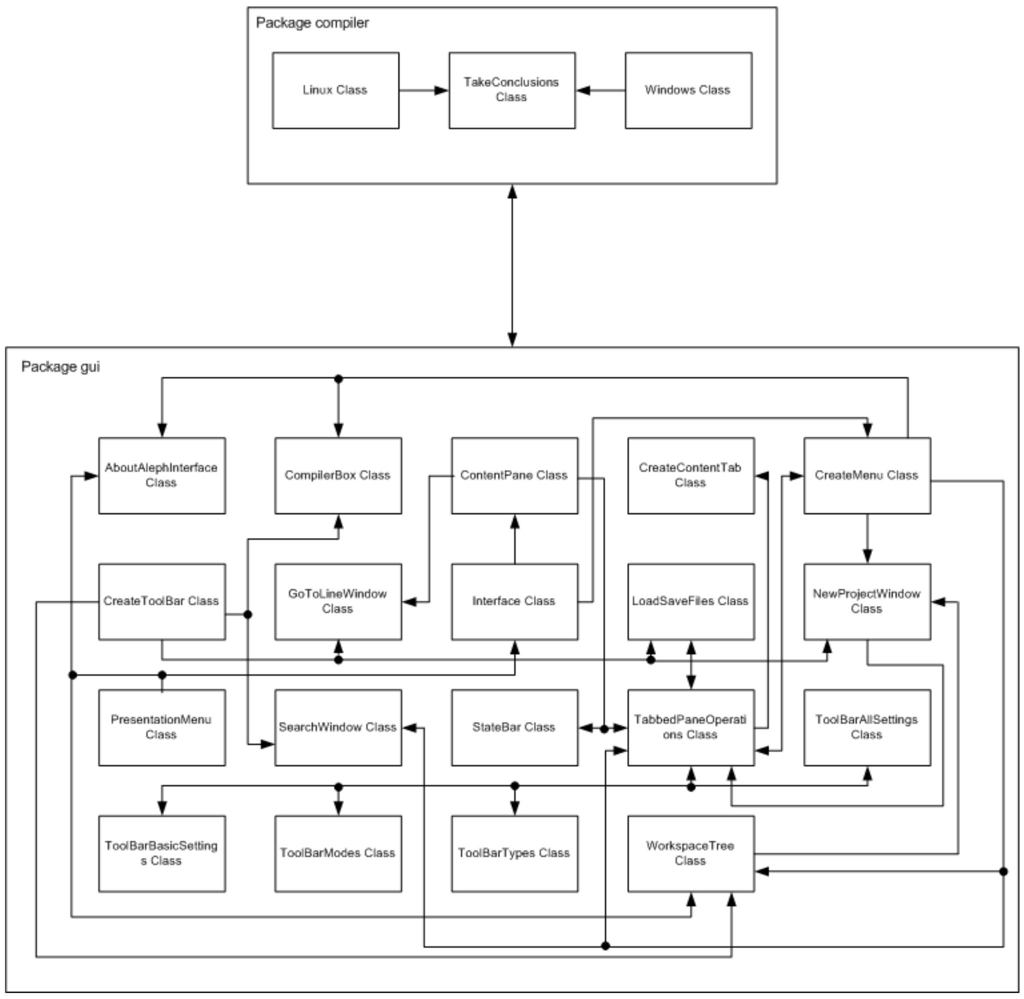


Figure 6.3: Aleph Interface Architecture.

In the Appendix B it's explained what which one of these classes do.

6.6 Test

The test of the Aleph Interface was a very important step in the development of this software. With this step the interface is now more robust making it more powerful and the chance of find an error has decreased. Many bugs were found, specially concerning in adding modes and how the results are shown to the user, although all founded bugs were successfully removed. It's important to mention that to run this interface it's necessary to run it in a directory without spaces. This problem is concerned by the the use of a Shell file or Bat file to call the YAP Prolog compiler. Many other solutions were tried to solve this problem but were not successful.

Chapter 7

Conclusions

7.1 Future Work

Future work on the Aleph Interface is related with the improvement of the system with additional functionalities for increasing its usability. Some of these functionalities are:

- the creation of a button or menu item which when pressed can organize the code of the file .b. For instance to put all settings, modes, determinations and types together;
- an option to check if there's any prolog error and tell to the user on which line that error was found;
- add a new tool bar tab to help the user to create or edit the files .n and .f;
- the development of the help contents to explain the functionalities of the Aleph Interface;

7.2 Final Considerations

All the proposed objectives for this project were successfully achieved. Now it's possible to create or edit models without prolog knowledge. These models can be saved and opened using the interface, and the results can be saved as well. The results are shown in a language very close to English to improve their perpection and allied to all these features, the interface can be used in Windows or Linux. With all these characteristics it's possible to conclude that this interface allows non ILP researchers to create the models, which was the first goal of this work.

Appendix A

Aleph Settings

This appendix shows all the usable settings that can be defined in the Aleph System and in the Aleph Interface as well. For each setting this appendix also inform which type of input should be inserted and gives a short explanation of what the setting does. The default values are also presented. All this information was taken from the Aleph Manual.

Setting: **abduce**
Type: boolean
Options: true or false
Default Value: undefined
Usability: set(abduce,+V)

If 'true' then abduction and subsequent generalisation of abduced atoms is performed within the induce loop. Only predicates declared to be abducible by abducible/1 are candidates for abduction.

Setting: **best**
Type: integer
Options: undefined
Default Value: undefined
Usability: set(best,+V)

V is a 'clause label' obtained from an earlier run. This is a list containing at least the number of positives covered, the number of negatives covered, and the length of a clause found on a previous search. Useful when performing searches iteratively.

Setting: **cache_clauselength**
Type: integer
Options: positive integer
Default Value: 3
Usability: set(cache_clauselength,+V)

Sets an upper bound on the length of clauses whose coverages are cached for future use.

Setting: **caching**
Type: boolean
Options: true or false
Default Value: false
Usability: set(caching,+V)

If 'true' then clauses and coverage are cached for future use. Only clauses up to length set by cache_clauselength are stored in the cache.

Setting: **check_redundant**
Type: boolean
Options: true or false
Default Value: false
Usability: set(check_redundant,+V)

Specifies whether a call to redundant/2 should be made for checking redundant literals in a clause.

Setting: **check_useless**
Type: boolean
Options: true or false
Default Value: false
Usability: set(check_useless,+V)

If set to 'true', removes literals in the bottom clause that do not contribute to establishing variable chains to output variables in the positive literal, or produce output variables that are not used by any other literal in the bottom clause.

Setting: **classes**
Type: list
Options: undefined

Default Value: undefined
Usability: set(classes,+V)

V is a list of classes to be predicted by the tree learner.

Setting: **clauselength**
Type: integer
Options: positive integer
Default Value: 4
Usability: set(clauselength,+V)

Sets upper bound on number of literals in an acceptable clause.

Setting: **clauselength_distribution**
Type: list
Options: undefined
Default Value: undefined
Usability: set(clauselength_distribution,+V)

V is a list of the form [p1-1,p2-2,...] where 'pi' represents the probability of drawing a clause with 'i' literals. Used by randomised search methods.

Setting: **clauses**
Type: integer
Options: positive integer
Default Value: undefined
Usability: set(clauses,+V)

Sets upper bound on the number of clauses in a theory when performing theory-level search.

Setting: **condition**
Type: boolean
Options: true or false
Default Value: false
Usability: set(condition,+V)

If 'true' then randomly generated examples are obtained after conditioning the stochastic generator with the positive examples.

Setting: **confidence**

Type: float number
Options: between 0.0 and 1.0
Default Value: 0.95
Usability: set(confidence,+V)

Determines the confidence for rule-pruning by the tree learner.

Setting: **construct_bottom**
Type: options
Options: saturation, reduction or false
Default Value: saturation
Usability: set(construct_bottom,+V)

Specifies the stage at which the bottom clause is constructed. If 'reduction' then it is constructed lazily during the search. This is useful if the bottom clause is too large to be constructed prior to search. This also sets the flag lazy_bottom to true. The user has to provide a refinement operator definition (using refine/2). If not, the refine parameter is set to auto. If 'false' then no bottom clause is constructed. The user would normally provide a refinement operator definition in this case.

Setting: **dependent**
Type: integer
Options: positive integer
Default Value: undefined
Usability: set(dependent,+V)

Denotes the argument of the dependent variable in the examples.

Setting: **depth**
Type: integer
Options: positive integer
Default Value: 10
Usability: set(depth,+V)

Sets an upper bound on the proof depth to which theorem-proving proceeds.

Setting: **explore**
Type: boolean
Options: true or false

Default Value: false
Usability: set(explore,+V)

If 'true' then forces search to continue until the point that all remaining elements in the search space are definitely worse than the current best element (normally, search would stop when it is certain that all remaining elements are no better than the current best. This is a weaker criterion). All internal pruning is turned off.

Setting: **evalfn**
Type: options
Options: coverage, compression, posonly, pbayes, accuracy, laplace, auto_m, mestimate, entropy, gini, sd, wracc, or user
Default Value: coverage
Usability: set(evalfn,+V)

Sets the evaluation function for a search.

Setting: **good**
Type: boolean
Options: true or false
Default Value: false
Usability: set(good,+V)

If 'true' then stores a Prolog encoding of 'good' clauses found in the search. A good clause is any clause with utility above that specified by the setting for minscore. If goodfile is set to some filename then this encoding is stored externally in that file.

Setting: **goodfile**
Type: filename
Options: undefined
Default Value: undefined
Usability: set(goodfile,+V)

Sets the filename for storing a Prolog encoding of good clauses found in searches conducted to date. Any existing file with this name will get appended.

Setting: **gsamplesize**
Type: integer

Options: positive integer
Default Value: 100
Usability: set(gsamplesize,+V)

The size of the randomly generated example set produced for learning from positive examples only.

Setting: **i**
Type: integer
Options: positive integer
Default Value: 2
Usability: set(i,+V)

Set upper bound on layers of new variables.

Setting: **interactive**
Type: boolean
Options: true or false
Default Value: false
Usability: set(interactive,+V)

If 'true' then constructs theories interactively with induce_rules and induce_tree.

Setting: **language**
Type: integer or inf
Options: integer greater or equal to 1 or inf
Default Value: inf
Usability: set(language,+V)

Specifies the number of occurences of a predicate symbol in any clause.

Setting: **lazy_on_contradiction**
Type: boolean
Options: true or false
Default Value: false
Usability: set(lazy_on_contradiction,+V)

Specifies if theorem-proving should proceed if a constraint is violated.

Setting: **lazy_on_cost**

Type: boolean
Options: true or false
Default Value: false
Usability: set(lazy_on_cost,+V)

Specifies if user-defined cost-statements require clause coverages to be evaluated. This is normally not user-set, and decided internally.

Setting: **lazy_negs**
Type: boolean
Options: true or false
Default Value: false
Usability: set(lazy_negs,+V)

If 'true' then theorem-proving on negative examples stops once bounds set by noise or minacc are violated.

Setting: **lookahead**
Type: integer
Options: positive integer
Default Value: false
Usability: set(lookahead,+V)

Sets a lookahead value for the automatic refinement operator (obtained by setting refine to auto).

Setting: **m**
Type: float number
Options: undefined
Default Value: undefined
Usability: set(m,+V)

Sets a value for 'm-estimate' calculations.

Setting: **max_abducibles**
Type: integer
Options: positive integer
Default Value: 2
Usability: set(max_abducibles,+V)

Sets an upper bound on the maximum number of ground atoms within any abductive explanation for an observation.

Setting: **max_features**
Type: integer
Options: positive integer or inf
Default Value: inf
Usability: set(max_features,+V)

Sets an upper bound on the maximum number of boolean features constructed by searching for good clauses.

Setting: **minacc**
Type: float number
Options: between 0.0 and 1.0
Default Value: 0.0
Usability: set(minacc,+V)

Set a lower bound on the minimum accuracy of an acceptable clause. The accuracy of a clause has the same meaning as precision: that is, it is p/(p+n) where p is the number of positive examples covered by the clause (the true positives) and n is the number of negative examples covered by the clause (the false positives).

Setting: **mingain**
Type: float number
Options: undefined
Default Value: 0.05
Usability: set(mingain,+V)

Specifies the minimum expected gain from splitting a leaf when constructing trees.

Setting: **minpos**
Type: integer
Options: positive integer
Default Value: 1
Usability: set(minpos,+V)

Set a lower bound on the number of positive examples to be covered by an acceptable clause. If the best clause covers positive examples below this number, then it is not added to the current theory. This can be used to prevent Aleph from adding ground unit clauses to the theory (by setting the value to 2). Beware: you can get counter-intuitive results in conjunction with the minscore setting.

Setting: **minposfrac**
Type: float number
Options: between 0.0 and 1.0
Default Value: 0.0
Usability: set(minposfrac,+V)

Set a lower bound on the positive examples covered by an acceptable clause as a fraction of the positive examples covered by the head of that clause. If the best clause has a ratio below this number, then it is not added to the current theory. Beware: you can get counter-intuitive results in conjunction with the minpos setting.

Setting: **minscore**
Type: float number
Options: undefined
Default Value: -inf
Usability: set(minscore,+V)

Set a lower bound on the utility of of an acceptable clause. When constructing clauses, If the best clause has utility below this number, then it is not added to the current theory. Beware: you can get counter-intuitive results in conjunction with the minpos setting.

Setting: **moves**
Type: integer
Options: greater or equal to 0
Default Value: undefined
Usability: set(moves,+V)

Set an upper bound on the number of moves allowed when performing a randomised local search. This only makes sense if search is set to rls and rls_type is set to an appropriate value.

Setting: **newvars**
Type: integer or inf
Options: positive integer or inf
Default Value: undefined

Usability: set(newvars,+V)

Set upper bound on the number of existential variables that can be introduced in the body of a clause.

Setting: **nodes**
Type: integer
Options: positive integer
Default Value: 5000
Usability: set(nodes,+V)

Set upper bound on the nodes to be explored when searching for an acceptable clause.

Setting: **noise**
Type: integer
Options: integer greater or equal to 0
Default Value: 0
Usability: set(noise,+V)

Set an upper bound on the number of negative examples allowed to be covered by an acceptable clause.

Setting: **nreduce_bottom**
Type: boolean
Options: true or false
Default Value: false
Usability: set(nreduce_bottom,+V)

If 'true' then removes literals in the body of the bottom clause using the negative examples.

Setting: **openlist**
Type: integer or inf
Options: integer greater or equal to 0 or inf
Default Value: inf
Usability: set(openlist,+V)

Set an upper bound on the beam-width to be used in a greedy search.

Setting: **optimise_clauses**

Type: boolean
Options: true or false
Default Value: false
Usability: set(optimise_clauses,+V)

If 'true' performs query optimisations.

Setting: **permute_bottom**
Type: boolean
Options: true or false
Default Value: false
Usability: set(permute_bottom,+V)

If 'true' randomly permutes literals in the body of the bottom clause, within the constraints imposed by the mode declarations.

Setting: **portray_examples**
Type: boolean
Options: true or false
Default Value: false
Usability: set(portray_examples,+V)

If 'true' executes goal aleph_portray(Term) where Term is one of train_pos, train_neg, test_pos, or test_neg when executing the command show(Term).

Setting: **portray_hypothesis**
Type: boolean
Options: true or false
Default Value: false
Usability: set(portray_hypothesis,+V)

If 'true' executes goal aleph_portray(hypothesis). This is to be written by the user.

Setting: **portray_literals**
Type: boolean
Options: true or false
Default Value: false
Usability: set(portray_literals,+V)

If 'true' executes goal aleph_portray(Literal) where Literal is some literal.

This is to be written by the user.

Setting: **portray_search**
Type: boolean
Options: true or false
Default Value: false
Usability: set(portray_search,+V)

If 'true' executes goal aleph_portray(search). This is to be written by the user.

Setting: **print**
Type: integer
Options: positive integer
Default Value: 4
Usability: set(print,+V)

Sets an upper bound on the maximum number of literals displayed on any one line of the trace.

Setting: **proof_strategy**
Type: options
Options: restricted_sld, sld, or user
Default Value: restricted_sld
Usability: set(proof_strategy,+V)

If 'restricted_sld', then examples covered are determined by forcing current hypothesised clause to be the first parent clause in a SLD resolution proof. If 'sld' then this restriction is not enforced. The former strategy is efficient, but not refutation complete. It is sufficient if all that is needed is to determine how many examples are covered by the current clause, which is what is needed when Aleph is used to construct a set of non-recursive clauses greedily (for example using the induce/0 command). If set to user then Aleph expects a user-defined predicate prove/2, the first argument of which is a clause C, and the second is an example E. prove(C,E) succeeds if example E is provable using clause C and the background knowledge.

Setting: **prooftime**
Type: integer or inf
Options: positive integer or inf
Default Value: restricted_sld

Usability: set(prooftime,+V)

Sets an upper bound on the time (in seconds) for testing whether an example is covered. Overrides any value set for searchtime.

Setting: **prune_tree**
Type: boolean
Options: true or false
Default Value: false
Usability: set(prune_tree,+V)

Determines whether rules constructed by the tree learner are subject to pessimistic pruning.

Setting: **record**
Type: boolean
Options: true or false
Default Value: false
Usability: set(record,+V)

If 'true' then trace of Aleph execution is written to a file. The filename is given by recordfile.

Setting: **recordfile**
Type: filename
Options: undefined
Default Value: undefined
Usability: set(recordfile,+V)

Sets the filename to write a trace of execution. Only makes sense if record is set to true.

Setting: **refine**
Type: options
Options: user, auto, or false
Default Value: false
Usability: set(refine,+V)

Specifies the nature of the customised refinement operator. In all cases, the resulting clauses are required to subsume the bottom clause, if one exists. If 'false' then no customisation is assumed and standard operation results.

If 'user' then the user specifies a domain-specific refinement operator with refine/2 statements. If 'auto' then an automatic enumeration of all clauses in the mode language is performed. The result is a breadth-first branch-and-bound search starting from the empty clause. This is useful if a bottom clause is either not constructed or is constructed lazily. No attempt is made to ensure any kind of optimality and the same clauses may result from several different refinement paths. Some rudimentary checking can be achieved by setting caching to true. The user has to ensure the following for refine is set to auto: (1) the setting to auto is done after the modes and determinations commands, as these are used to generate internally a set of clauses that allow enumeration of clauses in the language; (2) all arguments that are annotated as #T in the modes contain generative definitions for type T. These are called be the clauses generated internally to obtain the appropriate constants; and (3) the head mode is clearly specified using the modeh construct.

Setting: **resample**
Type: integer or inf
Options: integer greater or equal to 1 or inf
Default Value: 1
Usability: set(resample,+V)

Sets the number of times an example is resampled when selected by induce/0 or induce_cover/0. That is, is set to some integer N, then the example is repeatedly selected N times by induce/0 or induce_cover/0.

Setting: **rls_type**
Type: options
Options: gsat, wsat, rrr, or anneal
Default Value: undefined
Usability: set(rls_type,+V)

Sets the randomised search method. Requires search to be set to rls, and integer values for tries and moves.

Setting: **rulefile**
Type: filename
Options: undefined
Default Value: undefined
Usability: set(rulefile,+V)

Sets the filename for storing clauses found in theory (used by write_rules/0).

Setting: **samplesize**
Type: integer
Options: integer greater or equal to 0
Default Value: 0
Usability: set(samplesize,+V)

Sets number of examples selected randomly by the induce or induce_cover commands. The best clause from the sample is added to the theory. A value of 0 turns off random sampling, and the next uncovered example in order of appearance in the file of training examples is selected.

Setting: **scs_percentile**
Type: float
Options: greater than 0 and smaller or equal to 100
Default Value: undefined
Usability: set(scs_percentile,+V)

This denotes that any clause in the top V-percentile of clauses are considered 'good' when performing stochastic clause selection. Only meaningful if search is set to scs.

Setting: **scs_prob**
Type: float
Options: greater or equal to 0 and smaller than 1.0
Default Value: undefined
Usability: set(scs_prob,+V)

This denotes the minimum probability of obtaining a 'good' clause when performing stochastic clause selection. Only meaningful if search is set to scs.

Setting: **scs_sample**
Type: integer
Options: positive integer
Default Value: undefined
Usability: set(scs_sample,+V)

Determines the number of clauses randomly selected from the hypothesis space in a clause-level search. Only meaningful if search is set to scs. his overrules any samplesizes calculated from settings for scs_percentile and scs_prob.

Setting: **search**
Type: options
Options: bf, df, heuristic, ibs, ils, rls, scs id, ic, ar, or false
Default Value: bf
Usability: set(search,+V)

Sets the search strategy. If 'false' then no search is performed.

Setting: **searchtime**
Type: integer or inf
Options: integer greater or equal to 0 or inf
Default Value: inf
Usability: set(searchtime,+V)

Sets an upper bound on the time (in seconds) for a search.

Setting: **skolemvars**
Type: integer
Options: undefined
Default Value: 10000
Usability: set(skolemvars,+V)

Sets the counter for variables in non-ground positive examples. Each variable will be replaced by a skolem variable that has a unique number which is no smaller than V. This number has to be larger than the number of variables that would otherwise appear in a bottom clause.

Setting: **splitvars**
Type: boolean
Options: true or false
Default Value: false
Usability: set(splitvars,+V)

If set to 'true' before constructing a bottom clause, then variable co-references in the bottom clause are split apart by new variables. The new variables can occur at input or output positions of the head literal, and only at output positions in body literals. Equality literals between new and old variables are inserted into the bottom clause to maintain equivalence. It may also result in variable renamed versions of other literals being inserted into the bottom clause. All of this increases the search space considerably and can make the search explore redundant clauses. The current version also elects to perform variable splitting whilst constructing the bottom clause (in contrast to doing it dynamically whilst searching). This was to avoid unnecessary checks that could slow down the search when variable splitting was not required. This means the bottom clause can be extremely large, and the whole process is probably not very practical for large numbers of co-references. The procedure has not been rigourously tested to quantify this.

Setting: **stage**
Type: options
Options: saturation, reduction or command
Default Value: command
Usability: set(stage,+V)

Sets the stage of current execution. This is normally not user-set, and decided internally.

Setting: **store_bottom**
Type: boolean
Options: true or false
Default Value: false
Usability: set(store_bottom,+V)

Stores bottom clause constructed for an example for future re-use.

Setting: **subsample**
Type: boolean
Options: true or false
Default Value: false
Usability: set(subsample,+V)

If 'true' then uses a sample of the examples (set by value assigned to subsamplesize) to evaluate the utility of a clause.

Setting: **subsamplesize**
Type: integer or inf
Options: integer greater or equal to 1 or inf
Default Value: inf
Usability: set(subsamplesize,+V)

Sets an upper bound on the number of examples sampled to evaluate the utility of a clause.

Setting: **temperature**
Type: float
Options: non zero float number
Default Value: undefined
Usability: set(temperature,+V)

Sets the temperature for randomised search using annealing. Requires search to be set to rls and rls_type to be set to anneal.

Setting: **test_pos**
Type: filename
Options: filename or list of filenames
Default Value: undefined
Usability: set(test_pos,+V)

Sets the filename or list of filenames containing the positive examples for testing. No filename extensions are assumed and complete filenames have to be provided.

Setting: **test_neg**
Type: filename
Options: filename or list of filenames
Default Value: undefined
Usability: set(test_neg,+V)

Sets the filename or list of filenames containing the negative examples for testing. No filename extensions are assumed and complete filenames have to be provided.

Setting: **threads**
Type: integer
Options: integer greater or equal to 1
Default Value: 1
Usability: set(threads,+V)

This is experimental and should not be changed from the default value until further notice.

Setting: **train_pos**
Type: filename

Options: filename or list of filenames
Default Value: undefined
Usability: set(train_pos,-V)

Sets the filename or list of filenames containing the positive examples. If set, no filename extensions are assumed and complete filenames have to be provided. If not set, it is internally assigned a value after the read_all command.

Setting: **train_neg**
Type: filename
Options: filename or list of filenames
Default Value: undefined
Usability: set(train_neg,-V)

Sets the filename or list of filenames containing the negative examples. If set, no filename extensions are assumed and complete filenames have to be provided. If not set, it is internally assigned a value after the read_all command.

Setting: **tree_type**
Type: options
Options: classification, class_probability, regression, or model
Default Value: undefined
Usability: set(tree_type,+V)

Setting: **tries**
Type: integer
Options: positive integer
Default Value: undefined
Usability: set(tries,+V)

Sets the maximum number of restarts allowed for randomised search methods. This only makes sense if search is set to rls and rls_type is set to an appropriate value.

Setting: **typeoverlap**
Type: float
Options: greater than 0.0 and smaller or equal to 1.0
Default Value: undefined

Usability: set(typeoverlap,+V)

Used by induce_modes/0 to determine if a pair of different types should be given the same name.

Setting: **uniform_sample**
Type: boolean
Options: true or false
Default Value: false
Usability: set(uniform_sample,+V)

Used when drawing clauses randomly from the clause-space. If set set to 'true' then clauses are drawn by uniform random selection from the space of legal clauses. Since there are usually many more longer clauses than shorter ones, this will mean that clauses drawn randomly are more likely to be long ones. If set to 'false' then assumes a uniform distribution over clause lengths (up to the maximum length allowed by clauselength). This is not necessarily uniform over legal clauses. If random clause selection is done without a bottom clause for guidance then this parameter is set to false.

Setting: **updateback**
Type: boolean
Options: true or false
Default Value: true
Usability: set(updateback,+V)

If 'false' then clauses found by the induce family are not incorporated into the background. This is experimental.

Setting: **verbosity**
Type: integer
Options: integer greater or equal to 0
Default Value: 1
Usability: set(verbosity,+V)

Sets the level of verbosity. Also sets the parameter verbose to the same value. A value of 0 shows very little.

Setting: **version**
Type: undefined
Options: undefined

Default Value: undefined
Usability: set(version,-V)

V is the current version of Aleph. This is set internally.

Setting: **walk**
Type: float
Options: between 0 and 1
Default Value: undefined
Usability: set(walk,+V)

It represents the random walk probability for the Walksat algorithm.

Type: Parameter P and value V
Options: undefined
Default Value: undefined
Usability: set(+P,+V)

Sets any user-defined parameter P to value V. This is particularly useful when attaching notes with particular experiments, as all settings can be written to a file. For example, set(experiment,'Run 1 with background B0').

Appendix B

Javadoc

This appendix describes the function of each Java class of the Aleph Interface, the public functions of each class and the public variables. It's also described the entry arguments of each one and the return values. This information was taken from the exported Javadoc.

Package: compiler
Class: Linux

public class Linux
extends java.lang.Object

This class is responsible for:
- preparing the files to be read from the YAP Prolog compiler;
- run YAP using a SHELL file;
- delete the temporary files created for the compilation.

public Linux(java.lang.String selectedItem)
The constructor of the class Linux.
Parameters:

selectedItem - The project name which the user wants to run.

Package: compiler
Class: TakeConclusions

public class TakeConclusions
extends java.lang.Object

This class is responsible for write the conclusions done by the YAP Prolog compiler in a easier way to understand them.

public TakeConclusions(java.lang.String selectedItem, java.lang.String allText)
The constructor of the class TakeConclusions.
Parameters:

selectedItem - The project name which the user has chosen to run.

allText - The text which YAP produced for the selected project after running it.

Package: compiler
Class: Windows

public class Windows
extends java.lang.Object

This class is responsible for:
- preparing the files to be read from the YAP Prolog compiler;
- run YAP using a BAT file;
- delete the temporary files created for the compilation.

public Windows(java.lang.String selectedItem)
The constructor of the class Linux.
Parameters:

selectedItem - The project name which the user wants to run.

Package: gui
Class: AboutAlephInterface

public class AboutAlephInterface
extends javax.swing.JFrame

This class displays a window with some information about the Aleph Interface.

public AboutAlephInterface()

The constructor of the class AboutAlephInterface.

Package: gui
Class: CompilerBox

public class CompilerBox
extends javax.swing.JFrame

This class displays a window where the user can select one of the opened projects to run in the YAP Prolog compiler.

public CompilerBox()
The constructor of the class CompilerBox.

Package: gui
Class: CompilerBox

public class ContentPane
extends javax.swing.JPanel

This class is responsible for:
- create the Tabbed Panes for the inputs, outputs and tool bar;
- call the class which creates the State Bar.

public ContentPane()
The constructor for the class ContentPane.

public static javax.swing.JTabbedPane createTab
The Tabbed Pane for the inputs (files .b, .n, .f).

public static javax.swing.JTabbedPane createTabResults
The Tabbed Pane for the outputs after running the project in the YAP Prolog Compiler.

public static javax.swing.JTabbedPane createTabToolBar
The Tabbed Pane for the settings, modes and types.

Package: gui
Class: CreateContentTab

public class CreateContentTab
extends javax.swing.JPanel

This class is responsible to create the content of each Tabbed Pane of the inputs and the outputs. This content includes a Text Area and a Scroll Pane (if necessary). In the inputs it's also included another Text Area which shows the line numbers.

public CreateContentTab(java.lang.String name, java.lang.String data, int location)
The constructor of the class CreateContentTab.
Parameters:

name - The name of the file which will be placed in the title of the Tab.

data - The content to put on the Text Area.

location - The location distinguish an input tab from an output tab. If location it's equal to 1 it means it's an input. If it's equal to 2 it's an output.

public javax.swing.undo.UndoManager getUndoManager()
This function returns the object undoManager for the selected tab when it's called. Each tab of input has its own undoManager object. It's also useful to use this function to know if the project is already saved.
Returns:

The object undoManager.

Package: gui
Class: CreateMenu

public class CreateMenu
extends javax.swing.JMenuBar

This class is responsible for the creation of all the menu bar.

public CreateMenu()
The constructor of the class CreateMenu.

public javax.swing.JPopupMenu getJPopupTabInputs()
This function displays a Popup Menu when the user clicks with the third

button of the mouse in one input tab.
Returns:

The object which displays all the options.

public javax.swing.JPopupMenu getJPopupTabResults()
This function displays a Popup Menu when the user clicks with the third button of the mouse in one output tab.
Returns:

The object which display the save option.

Package: gui
Class: CreateToolBar

public class CreateToolBar
extends javax.swing.JToolBar

This class it's responsible for the creation and management of the Tool Bar.

public CreateToolBar()
The constructor of the class CreateToolBar.

public static javax.swing.JButton jSaveButton
Button used to save a project in the current workspace. It alternates from enabled/disabled according to the undo button. If undo button is enabled the save button is enabled as well. If undo button is disabled, the save button is also disabled.

Package: gui
Class: GoToLineWindow

public class GoToLineWindow
extends javax.swing.JFrame

This class displays a window where the user can go straight to a choosen line.

public GoToLineWindow()
The constructor of the class GoToLineWindow.

public static void viewActualLineNumber(int linenumber)

This function refresh the actual number of line in the gotoline window.
Parameters:

linenumber - The number of line where the caret position is.

public static void viewTotalNumberOfLines(int totallines)
This function refresh the total number of lines in the gotoline window.
Parameters:

totallines - The total number of lines of the selected input tab.

public void putVisible()
When the user wants to open this window, just put it visible.

Package: gui
Class: Interface

public class Interface
extends javax.swing.JFrame

This class is responsible for the creation and management of the Frame of the interface.

public Interface()
The constructor of the class Interface.

Package: gui
Class: LoadSaveFiles

public class LoadSaveFiles
extends javax.swing.JFileChooser

This class it's responsible for the management of save and load files.

public void setWorkspace(java.lang.String workspace)
This function refresh the workspace when the user change it in the Workspace window.
Parameters:

workspace - The directory to change the workspace.

public void loadManager(java.lang.String filter)
This function displays a file chooser to the user and sends the information of the files to be read to the class loadTheReceivedFile.
Parameters:

filter - The type of files which can be read.

public void loadTheReceivedFile(java.io.File selectedFile)
This function loads a file and shows its content in a new tab.
Parameters:

selectedFile - The name of the file to be load.

public void saveManager(int flagOpenSaveDialog) throws java.io.IOException
This function is responsible for the management of the input files to be saved.
Parameters:

flagOpenSaveDialog - This flag is used to know if it's necessary to display a file chooser for the user choose the directory where to save the project, or if it's just to save the project in the workspace.

Throws:

java.io.IOException - In case it's not possible to save in the choosen directory.

public void saveTheReceivedFile(java.lang.String nameOfFile) throws java.io.IOException
This function saves the project in the workspace and changes the title of the input tab for the choosen name of file.
Parameters:

nameOfFile - The name of the file to be saved.
Throws:

java.io.IOException - In case it's not possible to save in the choosen directory.

public void saveOutput() throws java.io.IOException
This function saves the selected output file.
Throws:

java.io.IOException - In case it's not possible to save in the choosen directory.

Package: gui
Class: NewProjectWindow

public class NewProjectWindow
extends javax.swing.JFrame

This class displays a window where the user can write the name of the new project.

public NewProjectWindow()
The constructor of the class NewProjectWindow.

Package: gui
Class: PresentationMenu

public class PresentationMenu
extends javax.swing.JFrame

This class it's responsible to show the presentation window which is displayed when the user runs the Aleph Interface.

public PresentationMenu()
The constructor of the PresentationMenu class.

public static void main(java.lang.String[] args)
The main method.

Package: gui
Class: SearchWindow

public class SearchWindow
extends javax.swing.JFrame

This class displays a window where the user can:
- find words in the text;
- replace the founded words for other word;
- replace all the founded words for other word.

public SearchWindow()
The constructor of the SearchWindow class.

public static java.util.List foundedWords
An array list with all the positions of the word that the user wants to find/replace.

public void findNext(int flagSearchWindow, java.lang.String textToFind)
This function is responsible for the management of find text in the selected input tab.
Parameters:

flagSearchWindow - This flag is used to know if the search was called from the search window or from the menu bar.

textToFind - The text which the user wants to search.

Package: gui
Class: StateBar

public class StateBar
extends javax.swing.JPanel

This class creates the State Bar and show to the user the actual and the total number of lines for the selected input tab.

public StateBar()
The constructor of the class StateBar.

public static void viewLineNumber(int lineNumber, int totalLines)
This function refreshes the actual and total number of lines.
Parameters:

lineNumber - The actual number of line of the selected input tab.

totalLines - The total number of lines of the selected input tab.

Package: gui
Class: TabbedPaneOperations

public class TabbedPaneOperations

extends java.lang.Object

This class is responsible for the management of all tabs in the interface. It includes:
- add new tabs;
- close a project, all tabs or just one tab;
- display option panels if the user close a project and haven't saved it yet;
- refresh the tool bar tabs when the user change from the input tabs.

public javax.swing.JTabbedPane addTabs(javax.swing.JTabbedPane jTabbedPane, java.lang.String name, java.lang.String description, java.lang.String data, int location)
This function is responsible for add a new tab for a Tabbed Pane.
Parameters:

jTabbedPane - The choosen Tabbed Pane. This tab can be to the tool bar, to the input or to the output.

name - The name of the tab.

description - The tool tip text to appear in the tab.

data - The content to be added in the tab.

location - The position where the tab should be added.

Returns:

The object Tabbed Pane.

public void closeTabInputIfNecessary()
This function close the Inputs tab (if exists) when it's called.

public void closeProjectManager()
This function is responsible for the management when the user wants to close a project. It includes:
- check if the project was already saved;
- create the tab Inputs if all projects were closed;
- create the tab Results if all projects were closed;
- refresh the tool bar.

public void closeAllTabsManager()
This function is responsible for the management when the user wants to close all tabs.

public int checkIfThereAreProjectsToBeSaved()
This function check if there are projects that weren't saved.
Returns:

If there's any file that wasn't saved returns 1, otherwise returns 0.

public javax.swing.JTextArea getEnabledTextArea()
This function is used to return the object text area of the selected input tab.
Returns:

The object of the selected input text area.

public javax.swing.JTextArea getEnabledTextAreaResults()
This function is used to return the object text area of the selected output tab.
Returns:

The object of the selected output text area.

public javax.swing.JTextArea getEnabledTextAreaLineNumbers()
This function is used to return the object text area with the line numbers of the selected input tab.
Returns:

The object of the selected input text area with the line numbers.

public javax.swing.JTextArea getTextAreaInCertainTab(int positionTab)
This function is used to return the object text area of a certain input tab.
Returns:

The object of the input text area in the position described in its argument.

Package: gui
Class: ToolBarAllSettings

public class ToolBarAllSettings
extends javax.swing.JPanel

This class is responsible for the creation and management of all Settings displayed in the tool bar and write them in the selected input tab.

public ToolBarAllSettings()
The constructor of the class ToolBarAllSettings.

Package: gui
Class: ToolBarBasicSettings

public class ToolBarBasicSettings
extends javax.swing.JPanel

This class is responsible for the creation and management of the basic Settings displayed in the tool bar and write them in the selected input tab.

public ToolBarBasicSettings()
The constructor of the class ToolBarBasicSettings.

Package: gui
Class: ToolBarModes

public class ToolBarModes
extends javax.swing.JPanel

This class is responsible for the creation and management of Modes displayed in the tool bar and write them in the selected input tab.

public ToolBarModes()
The constructor of the class ToolBarModes.

Package: gui
Class: ToolBarTypes

public class ToolBarTypes
extends javax.swing.JPanel

This class is responsible for the creation and management of Types displayed in the tool bar and write them in the selected input tab.

public ToolBarTypes()

The constructor of the class ToolBarTypes.

Package: gui
Class: WorkspaceTree

public class WorkspaceTree
extends javax.swing.JFrame

This class create and displays a tree with all available folders where the user can choose the workspace.

public WorkspaceTree(int flag, int root)
The constructor of the class WorkspaceTree.
Parameters:

flag - This flag is used to know if it's necessary to display the new project window to the user after choose the workspace.

root - To open the selected directory.

References

[1] Ian H.Witten and Eibe Frank, Data Mining Practical Machine Learning Tools and Techniques, 2nd Edition, 2005

[2] Bill Palace, Data Mining Overview, 1996 (http://www.anderson.ucla.edu/faculty/jason.frand/teacher/technologies/palace/index.htm)

[3] Jiawei Han and Micheline Kamber, Data Mining Concepts and Techniques, 2000

[4] Nils J. Nilsson, Introduction to Machine Learning, 1996

[5] Sašo Džeroski, Nada Lavrač, Inductive Logic Programming, Techniques and Applications, 1994

[6] Sašo Džeroski, Nada Lavrač, Relational Data Mining, Chapter 14 - Relational Data Mining Applications: An Overview

[7] David Hand, Heikki Mannila and Padhraic Smyth, Principles of Data Mining, 2001

[8] Tom M. Mitchell, Machine Learning, 1997

[9] Mariza Ferro, Aquisição de conhecimento de conjuntos de exemplos no formato atributo valor utilizando aprendizado de mquina relacional, July 2004 (http://www.teses.usp.br/teses/disponiveis/55/55134/tde-16112004-095938/)

[10] A. Srinivasan, The Aleph Manual (http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph_toc.html)

[11] Kurt Thearling, Information about data mining and analytic technologies
(http://www.thearling.com/text/dmwhite/dmwhite.htm)

[12] Ethem Alpaydin, Introduction to Machine Learning, October 2004

[13] Stephen H. Muggleton, Inductive Logic Programming
(http://www.doc.ic.ac.uk/∼shm/ilp.html)

[14] Lubomír Popelínský, On practical inductive logic programming, July 2000

[15] Rui Camacho, Extraco de Conhecimento 2007/2008, ILP
(http://paginas.fe.up.pt/∼ec/files_0708/slides/ilp.pdf)

[16] Patrick Blackburn, Johan Bos and Kristina Striegnitz, Learn Prolog Now, Februray 2003
(http://www.coli.uni-saarland.de/∼kris/learn-prolog-now/html/index.html)

[17] James Lu and Jerud J.Mead, Prolog a Tutorial Introduction
(http://www.soe.ucsc.edu/classes/cmps112/Spring03/languages/prolog/PrologIntro.pdf)

[18] Roman Bartak, Guide to Prolog Programming, 1998
(http://kti.mff.cuni.cz/∼bartak/prolog/contents.html)

[19] Yet Another Prolog (YAP)
(http://www.dcc.fc.up.pt/∼vsc/Yap/)

[20] David Wells, Object Services and Consulting, Inc., 1996
(http://www.objs.com/survey/wrap.htm)

[21] Vince Huston, OO design, Java, C++
(http://www.vincehuston.org/dp/adapter.html)