UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA
Dpto. de Tecnologías y Sistemas de Información

Bousi∼Prolog ver. 2.0: Implementation, User Manual and Applications

Autors: Pascual Julián-Iranzo and Juan Gallardo-Casero
Other contributors: Fernando Sáenz-Pérez (U. Complutense de Madrid)

This documentation was written and formatted with LaTeX. All the figures and diagrams that appear in it were designed with Gimp, Inkscape or ArgoUML. The shield of the cover is work of D. Villa, I. Díez, F. Moya, Xavigivax and G. Paula and is distributed under the Creative Commons Attribution 3.0 license.

ABSTRACT

Logic Programming is a programming paradigm based on first order logic that, in recent decades, has been used in fields such as knowledge representation, artificial intelligence or deductive databases. However, Logic Programming presents an important limitation when dealing with real-world problems where precise information is not available, since it does not have the tools to explicitly handle uncertainty and vagueness.

One of the related frameworks to this paradigm is Fuzzy Logic Programming, which integrates concepts coming from Fuzzy Logic into Logic Programming to solve the disadvantages of traditional logic languages when dealing with uncertainty and ambiguity.

In this work we propose a Fuzzy Logic Programming language based on similarity relations called Bousi$\sim$Prolog. The language Bousi$\sim$Prolog is an extension of Prolog that replaces the classic unification algorithm of the operational semantics of Logic Programming with a weak unification algorithm. The use of this algorithm allows easy handling of imprecise information, while making the query answering process more flexible.

The present work covers both the Bousi$\sim$Prolog language design and specification and its implementation. The system that implements the Bousi$\sim$Prolog language has been named BPL system and is composed of a command processor, a compiler and an interpreter.

Finally, in this work we also provide several practical examples in which the usefulness of Bousi$\sim$Prolog is highlighted to solve certain problems that could not be solved naturally with non-fuzzy logic languages.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODE LISTINGS

CHAPTER 1. INTRODUCTION, MOTIVATIONS AND GOALS

This chapter introduces the work that will be done in this work, justifies the need to carry it out and lists the main objectives that are to be achieved. At the end of the chapter, we explain how this documentation has been structured.

1.1. MOTIVATION

Since the end of the 1960s, Declarative Programming, and specifically Logic Programming, has been used in a wide range of applications, among which knowledge representation, artificial intelligence, deductive databases and metaprogramming stand out. Logic programming languages like Prolog [14] have reached a high degree of maturity in recent decades, thanks to which their use has extended beyond research and education.

However, because they are based on first order logic, the traditional logic programming languages lack mechanisms to deal explicitly with uncertainty or vagueness. This limitation is an obstacle to representing and managing many real-world problems where precise information is not available.

In this sense, it is widely accepted that fuzzy sets theory and fuzzy logic [44] constitute one of the mathematical basis for the treatment of vagueness and incomplete information. For this reason, in order to solve the aforementioned drawbacks, at the beginning of the 1980s Fuzzy Logic Programming arose, in which concepts of fuzzy logic were integrated within the core of pure logic programming.

Languages of this new paradigm are able to handle uncertainty, ambiguity and approximate reasoning in a natural way, so its use is applicable to fields such as fuzzy control and natural language processing.

At the present time there are different ways to make this integration between logic programming and fuzzy logic, but an approach that is especially useful, if what it is intended is to make the querying answering process more flexible, is the one presented in [39]. In the aforementioned article, the concept of similarity between terms is introduced at a syntactic level, and a new fuzzy operational mechanism is defined based on a variant of SLD resolution[1] called weak SLD resolution.

To see the usefulness of this new resolution strategy, consider the following simple argument inspired by the examples of the articles [39, 22]:

Thrillers are interesting.

«The Treasure Island» is an adventure novel.

For me, it is more or less true that thrillers and adventure novels are similar.

Therefore, it is more or less true that «The Treasure Island» is an interesting novel.

Formalizing the previous example in a traditional logic system is not trivial when intervening a relation and an intermediate truth degree («more or less true») which is characteristic of fuzzy logic. For this reason, it is necessary to implement into logic programming languages a fuzzy resolution strategy such as the WSLD resolution, which may consider terms, that are not syntactically equal, as similar and offer the possibility of assigning an approximation degree to these related terms.

1.2. MAIN GOAL

In this work we propose the design of a fuzzy logic programming language that uses the Weak SLD resolution strategy as an operational mechanism, and the construction of the necessary development tools to create, edit and execute programs written in that language .

The main objectives of this new language are to support flexible query answering, to allow the manipulation of fuzzy sets and to incorporate other features with which it is possible to easily handle imprecise information using declarative techniques.

The new language has been proposed as an extension of Prolog, the most commonly used logic programming language. So that it will maintain most of its syntactical features, finding the biggest differences in several specific constructions and in their resolution and unification algorithms.[2] For these reasons, the language has been baptized as Bousi∼Prolog (abbreviated BPL), with Bousi being the spanish acronym for «Unificación BOrrosa por SImilitud».

In Bousi∼Prolog the user will be able to declare within their programs relations of

[1]SLD resolution (acronym of «Linear Deduction with Selection function for defined clauses») is the operational mechanism on which a representative subset of logic programming is based.

[2]Unification is a process of great importance in logic programming by which, given two different terms, we calculate a substitution of variables that, when applied to these terms, makes them syntactically equal.

similarity between symbols using the operator $\sim$, as well as fuzzy sets using a special syntax. These relations and fuzzy sets will be taken into account during the answering search process to offer a wider range of solutions than would be obtained with a standard logic programming system.

Due to its characteristics, Bousi∼Prolog will be suitable for the construction of flexible deductive databases, knowledge-based systems, information retrieval systems and, in general, applications that require approximate reasoning.

Finally, it is important to note that this project has marked a turning point in the development of Bousi∼Prolog, having served to collect and document all the knowledge acquired so far, add the features that had been raised in recent months and get a stable version of this emerging programming language.

1.3. SPECIFIC GOALS

The main objectives that have been considered when designing and implementing the Bousi∼Prolog language, and that have marked the progress of the present work since its starting point, are the following:

1. Design a fuzzy logic programming language (Bousi∼Prolog) based on Weak SLD resolution that makes the search for answers more flexible and includes techniques for operating with fuzzy sets.

2. Implement the lexical, syntactic and semantic analysis of the programs written in the Bousi∼Prolog language.

3. Develop a compiler able of translating Bousi∼Prolog programs into intermediate structures that can be used later by a language interpreter.

4. Implement a Bousi∼Prolog interpreter that can resolve queries using the weak SLD resolution as the operational mechanism.

5. Build a command processor that allows loading and executing programs written in Bousi∼Prolog using the compiler and interpreter previously implemented.

6. Develop a user interface that facilitates both the use of the previous tools and the editing and maintenance of Bousi∼Prolog programs.

7. Apply the built system to various practical cases.

The Bousi∼Prolog command processor, the compiler and the interpreter, which as a whole from now on will be referred to as the Bousi∼Prolog system or BPL system, will be implemented in SWI-Prolog and to a lesser extent in C.

1.4. STRUCTURE OF THIS DOCUMENT

The documentation has been structured in 7 chapters whose contents are summarized below.

In Chapter 2 the antecedents and the state of the art regarding fuzzy logic programming is presented, including both an explanation of the theoretical concepts on which this paradigm is based and an overview of some of its main languages.

In Chapter 3 the characteristics of the Bousi∼Prolog language are described in detail and the approach that has been followed to translate the programs written in this language. Chapters 4 and **??** address the development of the Bousi∼Prolog system and the user interface, respectively, from requirement analysis to testing, through all the stages of Software Engineering.

Next, Chapter 5 shows a series of practical examples in which Bousi∼Prolog can be applied successfully, while in Chapter 6 the conclusions that have been extracted after the completion of this work and some proposals for future work are presented.

Finally, in the last part of this document we provide several appendices with the user and installation manuals of the implemented tools and the description of one of the notations used in this work.

CHAPTER 2. FUZZY LOGIC PROGRAMMING

In this chapter we explain how concepts from fuzzy logic can be integrated into classical logic programming, what leads to fuzzy logic programming.

The first sections of this chapter are intended to introduce various concepts of fuzzy logic, such as fuzzy relationships and linguistic variables. Subsequently, the characteristics of the approach to fuzzy logic programming used in the elaboration of this work are explained and the Weak SLD resolution is defined. Finally, a study of some existing programming languages within the fuzzy logic programming framework is carried out.

2.1. FUZZY LOGIC

Fuzzy logic is an extension of bivalent logic that emerged from the theory of fuzzy sets proposed by L. A. Zadeh in [44]. Its main characteristic is that the expressions will not only be true or false, but that they will be able to assign a truth value that oscillates between an absolute certainty and falsehood.

From its origins, the purpose of fuzzy logic has been to represent the uncertain and imprecise knowledge that is present in many problems of the real world, to make approximate reasoning about them. Thus, fuzzy logic offers mechanisms to deal with concepts and quantifiers of natural language that, due to their vagueness, would be difficult to represent with classical logic, such as «high», «thin», «far» , «cold», «little», «frequently», «more or less», etc. [46]

2.1.1. Fuzzy sets

Fuzzy set theory is a generalization of the classical set theory in which the limit that determines whether an element belongs to a set or not is not completely defined, so it is considered the case that an element belongs partially to a set [36]

Formally, one of the ways to define a classic set A on a domain $\mathcal{U}$ is to specify its membership function or *characteristic function* $\mu_A(x)$. This function can only take the values

0 or 1 depending on whether the element x belongs to the set or not.

$$A = \{x \mid x \in \mathcal{U} \wedge \mu_A(x) = 1\}$$

$$\mu_A(x) = \begin{cases} 1 \text{ si } x \in A \\ 0 \text{ si } x \notin A \end{cases}$$

In fuzzy logic, the characteristic function $\mu(x)$ of a fuzzy set can take any value within the $[0, 1]$ interval. Therefore, *fuzzy sets* are defined by tuples consisting of an element x and the value of the characteristic function applied to that element.

$$A' = \{(x, \mu_{A'}(x)) \mid x \in \mathcal{U}\}$$

$$\mu_{A'}(x) \in [0, 1]$$

The characteristic functions of fuzzy sets can have different forms depending on the domain considered, the complexity of the fuzzy set and the precision that is necessary. Because of its easy definition and simplicity when making calculations, the membership functions most commonly used when dealing with numerical domains are the functions: *singleton*, triangular, trapezoidal, pseudo-exponential, *gaussiana*, Γ(gamma) and S [35]. The Figure 2.1 shows the graphs of five of these characteristic functions.

2.1.2. Fuzzy Relations

In general terms, a relation is a set of tuples that allow representing the association or connection between the elements of two or more sets (not necessarily different). Depending on the number of sets involved in a relation, it is called binary, ternary, quaternary or, in general, n-ary relation.

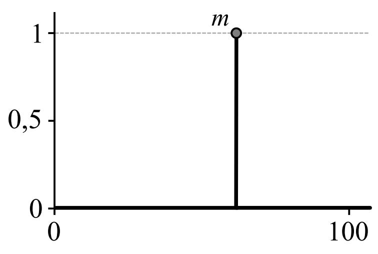
Formally, given a set of sets $X_1, \ldots, X_n$, a relation is any subset of the Cartesian product $X_1 \times \cdots \times X_n$. Therefore, being actually sets, relationships can also be characterized by membership functions. In particular, a binary relation of a set X can be defined in the following way:

$$R = \{(u, v) \mid u, v \in X\}$$

$$\mu_R(u, v) = \begin{cases} 1 \text{ si } (u, v) \in R \\ 0 \text{ si } (u, v) \notin R \end{cases}$$

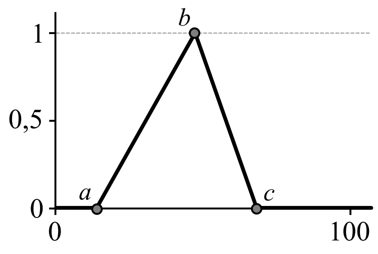
Extending the above definition to the fuzzy logic case, a *fuzzy binary relation* on a set X can be expressed as a fuzzy subset on the product $X \times X$, and represents the degree of

Domain point (*singleton*)



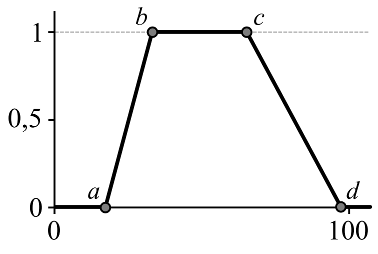
$$\mu(x) = \begin{cases} 1 & \text{if } x = m \\ 0 & \text{if } x \neq m \end{cases}$$

Triangular function



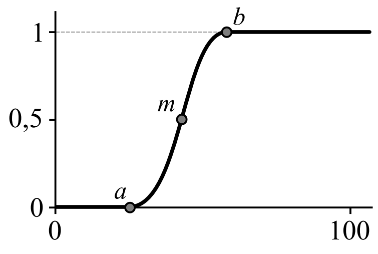
$$\mu(x) = \begin{cases} 0 & \text{if } x < a \\ \dfrac{x-a}{b-a} & \text{if } a \leq x < b \\ \dfrac{c-x}{c-b} & \text{if } b \leq x < c \\ 0 & \text{if } x \geq c \end{cases}$$

Trapezoidal function



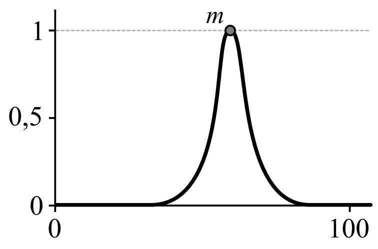
$$\mu(x) = \begin{cases} 0 & \text{if } x < a \\ \dfrac{x-a}{b-a} & \text{if } a \leq x < b \\ 1 & \text{if } b \leq x < c \\ \dfrac{d-x}{d-c} & \text{if } c \leq x < d \\ 0 & \text{if } x \geq d \end{cases}$$

S function



$$\mu(x) = \begin{cases} 0 & \text{if } x < a \\ 2\left(\dfrac{x-a}{b-a}\right)^2 & \text{if } a \leq x < m \\ 1 - 2\left(\dfrac{x-b}{b-a}\right)^2 & \text{if } m \leq x < b \\ 1 & \text{if } x \geq b \end{cases}$$

Pseudo-exponential function



$$\mu(x) = \frac{1}{1 + k(x-m)^2}$$

donde $k > 1$

Figure 2.1: Graphs and definitions of several characteristic functions [35].

Reflexivity	**Symmetry**	**Transitivity**	**Name**
✗	✗	✓	Strict order relation
✓	✓	–	Proximity Relation
✓	✗	✓	Partial order relation
✓	✓	✓	Similarity relation[1]

Legend: ✓ the property must be fulfilled for all $r \in R$.
✗ the property must not be fulfilled for any $r \in R$.
– the property can be fulfilled for some $r \in R$.

Table 2.1: Properties of some common relations.

presence or absence of interconnection between the elements of X [36]. As fuzzy subsets, the characteristic functions of fuzzy relations can take any value between 0 and 1.

$$R' = \{((u, v), \mu_{R'}(u, v)) \mid u, v \in X\}$$
$$\mu_{R'}(u, v) \in [0, 1]$$

Fuzzy relations, as classical relations, can have a series of properties, among which the following stand out:

- **Reflexivity**: $R(x, x) = 1$ for all $x \in X$.
- **Symmetry**: $R(x, y) = R(y, x)$ for all $x, y \in X$.
- **Transitivity**: $R(x, z) \geq R(x, y) \bigtriangleup R(y, z)$ for all $x, y, z \in X$.

The combination of these properties gives rise to different kinds of relations, such as those shown in Table 2.1.

On the other hand, the $\bigtriangleup$ operator that appears in the transitivity definition represents a *t-norm*, an extension of the *boolean* operator *AND* to the fuzzy logic. Some of the most common t-norms in the literature are [31]:

- Minim / Gödel: $x \bigtriangleup y = \min(x, y)$.
- Product: $x \bigtriangleup y = xy$.
- Łukasiewicz: $x \bigtriangleup y = \max(0, x + y - 1)$.
- Drastic intersection: $x \bigtriangleup y = \begin{cases} \min(a, b) & \text{if } a = 1 \text{ o } b = 1 \\ 0 & \text{otherwise} \end{cases}$
- Hamacher's product: $x \bigtriangleup y = \begin{cases} 0 & \text{if } a = b = 0 \\ \dfrac{xy}{x + y - xy} & \text{otherwise} \end{cases}$

[1]The term «similarity relation» is often used with preference to the term «equivalence relation» when referring to fuzzy relations.

extremely	very	more or less	somewhat
$h(x) = x^3$	$h(x) = x^2$	$h(x) = \sqrt{x}$	$h(x) = \sqrt[3]{x}$

Table 2.2: Definition of several linguistic modifiers [19].

2.1.3. Linguistic variables

A *linguistic variable* is a variable whose allowed values are words or sentences of a certain language instead of quantitative terms [45]. For example, «temperature» would be a linguistic variable if it were defined through natural language words such as «very cold», «cold», «warm», «not hot», «hot» or «very hot», in place with concrete values such as «18 C» or «–20 F».

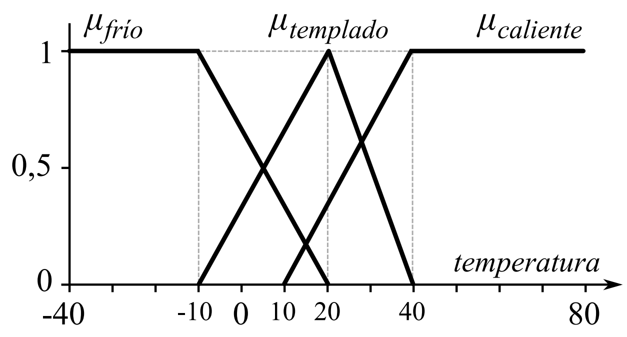
According to Zadeh [45] a linguistic variable $\mathcal{V}$ is characterized by a tuple of five elements $\langle X, T(X), U, G, M \rangle$, where:

- X is the name of the linguistic variable.
- $T(X)$ is the set of terms or linguistic values that the variable can take.
- U represents the universe of discourse.
- G is a grammar that contains the rules to generate the terms of $T(X)$.
- M is a semantic rule that relates each linguistic value x of $T(X)$ with its meaning, $M(x)$, expressed by means of a fuzzy subset of U[2]. In turn, these fuzzy subsets are characterized by membership functions μ_x that assign to each u value of the universe of discourse their membership in $M(x)$.

As an example, Figure 2.2 shows a possible definition of the linguistic variable «temperature». In this example only three very simple linguistic values have been considered («cold», «warm» and «hot»), but other more complex ones could have been specified such as «very cold», «little hot» or «more or less warm».

In order to facilitate the definition of this type of complex linguistic terms, composed from other simpler terms, the concept of *linguistic modifier* is introduced. A linguistic modifier is a function $[0, 1] \longrightarrow [0, 1]$ which is applied to the result of the characteristic function of a linguistic term in order to alter its meaning. Table 2.2 presents four linguistic modifiers of the many that can be defined.

[2]Given a universe of discourse U, $\mathcal{F}(U)$ denotes the set of all fuzzy subsets of U. Thus, the semantic rule M associates an element of $\mathcal{F}(U)$ with each linguistic value of $T(X)$.



$$
\begin{aligned}
&X = \text{«temperature»}\\
&T(X) = \{cold, hot, warm\}\\
&U = [-40, 80] \text{ Celsius degrees}\\
&M : T(X) \longrightarrow \mathcal{F}(U)\\
&\qquad cold \mapsto \mu_{cold}\\
&\qquad hot \mapsto \mu_{hot}\\
&\qquad warm \mapsto \mu_{warm}
\end{aligned}
$$

Figure 2.2: Example of definition of the linguistic variable «temperature».

2.1.4. Fuzzy relations and linguistic variables

There are several proposals to integrate fuzzy sets and linguistic variables into fuzzy logic programming languages, as can be seen in [41, 37]. This section focuses on the proposal of [23, 19], which consists in transforming the information that is represented with a linguistic variable into a reflexive fuzzy relation.

The main strength of this approach is that it distinguishes between specific and general knowledge. An example of the importance of this distinction can be seen with the linguistic terms «young» and «between 17 and 20 years old»: while a person who is between 17 and 20 years old is clearly young, a young person must not necessarily be between 17 and 20 years old. For this reason, the relationship between the values «between 17 and 20 years old» and «young» and between the values «young» and «between 17 and 20 years old» should not be the same, obtaining thus a fuzzy relation that satisfies the reflexive property but not the symmetric one[3].

Given a linguistic variable $\mathcal{V} = \langle X, T(X), U, G, M\rangle$, you can define a binary fuzzy relation R on the set $T(X)$ that for each pair of terms of $T(X)$ it assigns a degree of interconnection between 0 and 1.

$$
\begin{aligned}
R : T(X) \times T(X) &\longrightarrow [0, 1]\\
(T_1, T_2) \quad &\mapsto \quad \alpha
\end{aligned}
$$

The method proposed by [23, 19] to calculate the relationship between the different terms of a linguistic variable distinguishes the following cases:

a) If the second linguistic term is defined by a fuzzy domain point or *singleton* (see Figure 2.1), the relationship must be calculated directly using the membership function

[3]Actually, the obtained relation is symmetric but only at the level of its connections, that is, for any values â€‹â€‹a and b, if $R(a, b) \neq 0$ then $R(b, a) \neq 0$ but it does not have to be fulfilled that $R(a, b) = R(b, a)$.

of the other term. Thus, given two linguistic terms T_1 and T_2, if the meaning of T_2 is represented with a fuzzy domain point, the relationship between both values would be:

$$R(T_1, T_2) = \mu_{T_1}(t_2)$$

where t_2 is the only value for which $\mu_{T_2}(t_2) = 1$

b) Otherwise, a technique known as *matching* is applied to calculate the similarity between the fuzzy subsets of both terms. A *matching* function that has already been used successfully in systems like FuzzyCLIPS [33] is the following, extracted from [7]:

$$R(T_1, T_2) = \begin{cases} P(T_1, T_2) & \text{if } N(T_1, T_2) > 0.5 \\ (N(T_1, T_2) + 0.5) * P(T_1, T_2) & \text{otherwise} \end{cases}$$

where $P(T_1, T_2) = \max\left(\min\left(\mu_{T_1}(u), \mu_{T_2}(u)\right)\right) \quad \forall u \in U$

and $N(T_1, T_2) = 1 - P(\bar{T}_1, T_2)$,

being $\bar{T}_1$ the complement of the term T_1, whose membership function is $\mu_{\bar{T}_1}(u) = 1 - \mu_{T_1}(u) \quad \forall u \in U$

2.2. FOUNDATIONS OF FUZZY LOGIC PROGRAMMING

As was mentioned, classical Logic programming languages do not have mechanisms to handle uncertainty or inaccurate information, which represents a limitation when dealing with certain real-world problems.

Fuzzy logic programming attempts to mitigate these problems by introducing concepts from fuzzy logic, such as those studied in section 2.1, into the core of a logic programming language.

However, as indicated in [38], currently there is no standard way of integrating fuzzy logic and logic programming, but there are several lines of work in that direction, from those that replace the classic resolution mechanism by a fuzzy resolution strategy [30, 43, 12] to those that extend SLD resolution with a fuzzy unification algorithm [10, 26, 39], going through hybrid approaches like [3, 42]. Other frameworks related to fuzzy logic programming are the qualified logic programming [6] and the generalized annotated logical programming [25].

Of all mentioned proposals, an approach that is especially useful if what is intended with the language is to make the search for solutions more flexible is the one that M. Sessa introduces in [39], which is based on a variant of the SLD resolution strategy called: similarity-based SLD resolution. From now on, this strategy will be named as the Weak SLD resolution or WSLD resolution.

2.2.1. Goals of a fuzzy logic programming system

Roughly speaking, the objective of a logic programming system is to check if a goal $\mathcal{Q}$ is derivable from a program Π and to know for which values of the goal variables this condition will be fulfilled.

In fuzzy logic programming, since intermediate truth degrees between absolute certainty and falsehood must be considered, it is also interesting to know to what extent the goal $\mathcal{Q}$ is derivable from Π. To do this, in the way proposed by [39], the concept of *approximation degree* is introduced in the processes of unification and resolution, based on a similarity relation (i.e., a reflexive, symmetric and transitive relation) between the constant symbols, the function symbols and the relation symbols of the program alphabet.

For example, given the program $\Pi = \{p(a)\}$ and the relation $R(a, b) = 0.5$, for a goal like $\mathcal{Q} = p(b)$ a fuzzy logic system should answer that $p(b)$ is true with approximation degree 0.5, since $p(a)$ is a fact of the program and a is similar to b with degree 0.5.

2.2.2. Weak unification

The WSLD resolution strategy replaces the classic unification algorithm of Martelli and Montanari [28] by an extension of it that we name *weak unification algorithm.*

In this variant of the unification algorithm two expressions $\mathcal{E}_1 = f(t_1, \ldots, t_n)$ and $\mathcal{E}_2 = g(s_1, \ldots, s_n)$ are unifiable if f and g are similar to each other with a degree greater than a certain cut value $\lambda > 0$ and each of the arguments t_i and s_i pairwise weakly unify. Note that now the symbols f and g do not have to be syntactically equal so that the expressions can be unified, but it is enough that they are considered similar.

The weak unification algorithm uses a state transition system analogous to the classical algorithm, but adds a new element to the concept of state: the approximation degree. In this way, the initial state, $\langle\{\mathcal{E}_1 \approx \mathcal{E}_2\}, id, 1\rangle$, is formed by the initial unification problem, the identity substitution and the maximum approximation degree. The objective is to apply the

Regla 1: Term decomposition
$\dfrac{\langle\{f(t_1,\ldots,t_n) \approx g(s_1,\ldots,s_n)\} \cup E, \theta, \alpha\rangle, R(f,g) = \beta \geq \lambda}{\langle\{t_1 \approx s_1,\ldots,t_n \approx s_n\} \cup E, \theta, \min(\alpha,\beta)\rangle}$ where f and g can be either function symbols or relation symbols, or constants if $n = 0$; λ is the cut value
Regla 2: Removal of trivial equations.
$\dfrac{\langle\{X \approx X\} \cup E, \theta, \alpha\rangle}{\langle E, \theta, \alpha\rangle}$ where X is any variable
Regla 3: Swap
$\dfrac{\langle\{t \approx X\} \cup E, \theta, \alpha\rangle}{\langle\{X \approx t\} \cup E, \theta, \alpha\rangle}$ if the term t is not a variable
Regla 4: Variable elimination
$\dfrac{\langle\{X \approx t\} \cup E, \theta, \alpha\rangle}{\langle\{X/t\}(E), \{X/t\} \circ \theta, \alpha\rangle}$ if t is a term in which the variable X does not occur
Regla 5: Failure rule
$\dfrac{\langle\{f(t_1,\ldots,t_n) \approx g(s_1,\ldots,s_n)\} \cup E, \theta, \alpha\rangle, R(f,g) < \lambda}{\langle Failure, \theta, \alpha\rangle}$ where f and g can be either function or relation symbols
Regla 6: Occur check)
$\dfrac{\langle\{X \approx t\} \cup E, \theta, \alpha\rangle}{\langle Failure, \theta, \alpha\rangle}$ if the variable X occurs in the term t

Table 2.3: Transition rules of the weak unification algorithm [39, 17].

rules of Table 2.3 until reaching one of the two final states:

- Success state: $\langle\emptyset, \theta, \alpha\rangle$. Then, the expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ are unifiable with approximation degree α and θ is the weak most general unifier of both expressions (this weak most general unifier, contrary to what happens in the classical case, is not unique).
- Failure state: $\langle Failure, \theta, \alpha\rangle$. Then, the expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ can not be unified.

The only rules of Table 2.3 that have a behavior different from the classical case are the first and the fifth. Rule 1 is the one that contributes to calculate the final approximation degree of the initial expressions, always taking the minimum of all the relation values that are involved in the unification process. On the other hand, Rule 5 has had to be adapted to take into account the case where the function or relation symbols rooting an expression (term or atomic formula) are neither syntactically equal to nor similar to each other.

2.2.3. WSLD Resolution

The WSLD resolution inference rule is like the SLD resolution rule in classical logic programming, except for the use of the weak unification algorithm and the incorporation of the approximation degrees. During a WSLD resolution process, it is assigned an approximation degree to each goal clause, that is calculated from the degrees resulting from the weak unification steps.

Initially, the goal clause to be resolved has associated the top approximation degree (1). Then, as the WSLD resolution steps are performed, this degree can progressively decrease if in the unification of the complementary literals of the resolved clauses, the confronted symbols, related by a similarity relation, have values which are smaller than the previous degree obtained up to that moment.

When the result of a WSLD resolution step is the empty clause, $\Box$, the minimum of the approximation degrees obtained in that step and the previous steps of the WSLD derivation will be the approximation degree of the computed answer, that is, the degree in which $\Pi \vdash \theta(\mathcal{Q})$ is met.

Based on the definition of the classical SLD resolution step, a *WSLD resolution step* can be expressed as follows:

$$\frac{\begin{array}{l}\mathcal{G} \equiv \leftarrow \mathcal{A}_1 \wedge \cdots \wedge \mathcal{A}_{j-1} \wedge \boxed{\mathcal{A}_j} \wedge \mathcal{A}_{j+1} \wedge \cdots \mathcal{A}_n \quad \text{with degree } \alpha \\ \mathcal{C} \equiv \boxed{\mathcal{H}} \leftarrow \mathcal{B}_1 \wedge \cdots \wedge \mathcal{B}_m\end{array}}{\mathcal{G}' \equiv \leftarrow \theta(\mathcal{A}_1 \wedge \cdots \wedge \mathcal{A}_{j-1} \wedge \mathcal{B}_1 \wedge \cdots \wedge \mathcal{B}_m \wedge \mathcal{A}_{j+1} \wedge \cdots \wedge \mathcal{A}_n) \quad \text{with degree } \min(\alpha, \beta)}$$

where $\mathcal{G}$ is the goal clause that you want to solve,

$\mathcal{C}$ it is a variant of one of the program clauses,

θ is a weak most general unifier of the literals $\mathcal{H}$ and $\mathcal{A}_j$

y β is the approximation degree resulting from the weak unification of $\mathcal{H}$ and $\mathcal{A}_j$

2.3. FUZZY LOGIC PROGRAMMING SYSTEMS

Since the birth of fuzzy logic programming, in the early 70s, several ways of incorporating the concepts of fuzzy logic into logic programming languages have been proposed and put into practice.

In this section, several fuzzy logic programming systems are reviewed that use unification and resolution mechanisms based on similarity relations, which is the same approach that it is intended to be followed in this project.

2.3.1. Bousi∼Prolog (low-level implementation)

Within the Dec-tau research group of the University of Castilla-La Mancha, two different lines of work are maintained around the language Bousi∼Prolog[4].

The first line, which is known as *high-level implementation*, consists of creating a compiler that translates Bousi∼Prolog programs to conventional Prolog programs, and then executes them through an interpreter of that language. This is the approach that has been followed for the realization of the present work.

On the other hand, the second line of work, usually referred to as *low-level implementation*, is based on designing an extension of Warren's abstract machine (WAM) that is able of handling fuzzy relations. The goal is to execute the Bousi∼Prolog programs directly on a WAM without first translating them into the Prolog language. The work of the Dec-tau group on this line can be found in [17, 19, 18, 38].

As will be explained in more detail later, the syntax of Bousi∼Prolog is essentially the same as that of Prolog but incorporates a new predefined operator, $\sim$. This operator allows you to declare the entries of a similarity or proximity relation between symbols of the alphabet in the following way:

```
<symbol_1> ~ <symbol_2> = <approximation_degree>.
```

Sentences like the previous one are called *similarity equations* or, more generally, *proximity equations*. An equation of the type «$p \sim q = \alpha$» should be understood as «p *is similar to* q *with degree* α » but also in the opposite direction, «q *is similar to* p *with degree* α». Therefore, it actually defines two entries in the relation: $R(p, q) = \alpha$ and its symmetric $R(q, p) = \alpha$.

[4]http://dectau.uclm.es/bousi-prolog/

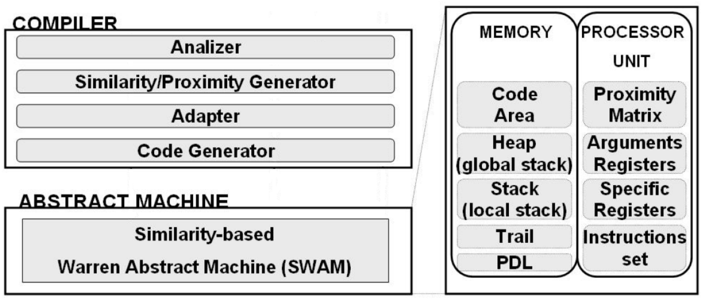


Figure 2.3: Structure of the low level implementation of Bousi∼Prolog [18].

Starting from a Bousi∼Prolog program composed of rules, facts, directives and proximity equations, the compiler of the low level implementation of Bousi∼Prolog generates machine code for the extended Warren's abstract machine, that we call *Similarity WAM* (SWAM). To do this, it follows the four stages shown in Figure 2.3, which are explained below [16, 18]:

a) **Analyzer**. It is responsible for performing the lexical and syntactic analysis of the input program, while generating an internal representation of the source code.

b) **Similarity / proximity Generator**. Using the proximity equations declared in the program, the reflexive, symmetric and, (optionally) transitive closure of the specified relation is generated. The result is a similarity or proximity relation between the symbols of the alphabet that is stored in a internal memory (called *proximity matrix*) of the Processor Unit in the SWAM to be used later[5].

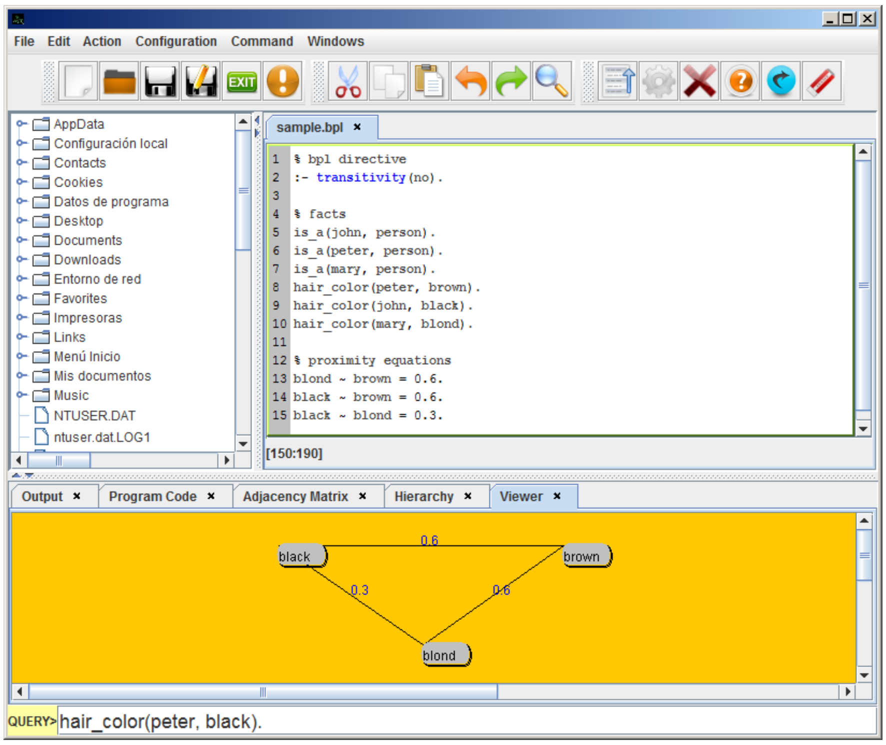
c) **Adapter**. From the source code of the program and the complete fuzzy relation generated in the previous step, the system internally constructs a new program in an extended language where approximation degrees may appear in the body of the clauses.

In short, each clause of the program is duplicated once for each symbol that is similar to rooted symbol in its head, replacing it with the similar symbol and adding the approximation degree between them at the beginning of the new clause. Observe the following example to understand the process that takes place in this phase:

```
a ~ b = 0.5.    ⇒    a :- 1.0, true.
a :- true.           b :- 0.5, true.
```

d) **Code Generator**. Taking as input the transformed program obtained in the previous

[5]In this phase, the fuzzy sets defined in the program with the help of the special directives `domain` and `fuzzy_set`, are also transformed into a fuzzy relation, following the techniques explained in section 2.1.4.



```
% bpl directive
:- transitivity(no).

% facts
is_a(john, person).
is_a(peter, person).
is_a(mary, person).
hair_color(peter, brown).
hair_color(john, black).
hair_color(mary, blond).

% proximity equations
blond ~ brown = 0.6.
black ~ brown = 0.6.
black ~ blond = 0.3.
```

```
hair_color(peter, black).
```

Figure 2.4: Unicorn graphic environment for the Bousi∼Prolog language.

phase, the machine code for the SWAM is generated, and it is stored in a specific memory area: the *code area* which is destined to this end.

The SWAM instruction set includes a series of specific machine instructions to manipulate the similarity degrees and the fuzzy unification processes that are not found in a standard WAM as described in [2]. In addition, the SWAM introduces several new records into the internal structures of the WAM to store the similarity degree computed during the WSLD resolution.

To interact with the compiler and the SWAM of the Bousi∼Prolog low level implementation, a graphical user interface implemented in Java, called Unicorn, has been created. The main panel window is shown in Figure 2.4. The Unicorn environment allows, among others, the following tasks:

- Edit, load, save and compile programs
- Launch queries to a program to see the answers returned by the system. From each one of them is shown the computed answer substitution and the approximation degree.
- See a graphical representation of the similarity or proximity relation defined in the program.

- Examine the SWAM machine code generated by the compiler.
- Analyze step by step what instructions have been executed during the resolution of a query.

The low-level implementation of Bousi$\sim$Prolog is quite efficient at runtime because the source programs are compiled into WAM instructions, which is the same thing that most Prolog-based logic programming systems do. Likewise, it is compatible with several of the extra logic features of Prolog, as the cut operator or input/output predicates.

Bousi$\sim$Prolog and, in particular, its low-level implementation, can serve to solve various types of problems in which imprecise or uncertain information needs to be handled. In [38] several examples of language use applied to the fields of deductive databases,[6]knowledge based systems, information retrieval and, in general, approximate reasoning are presented.

2.3.2. SiLog

SiLog [26, 27] is a Prolog interpreter developed in Java that implements weak unification and WSLD resolution. It is an extension of an existing logic programming environment called W-Prolog,[7]also based on Java.

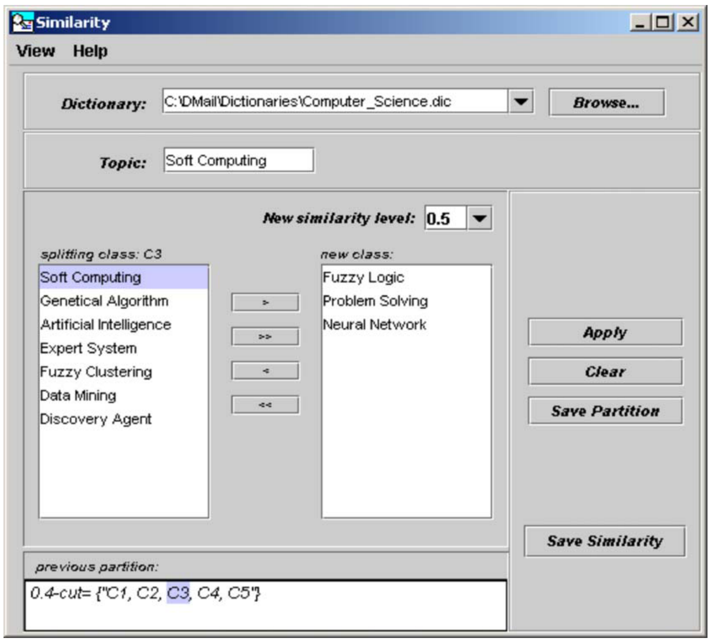
Unlike other fuzzy logic programming systems, where the definition of the programs and the similarity relations is carried out at the same time, the interaction with the SiLog system is divided into two clearly differentiated phases. In the first place, the similarity relation that will intervene in fuzzy unification is constructed, and in the second place the Prolog program is loaded and the queries that wish to be solved are introduced.

To facilitate the definition of fuzzy relations, SiLog has a panel as the one seen in Figure 2.5. From a text file with a list of initially isolated terms, the programmer can group those terms that he considers similar in sets named λ_i-cut. Each λ_i-cut is formed by one or several equivalence classes in which terms related to each other with the same similarity degree are grouped. This degree of similarity is calculated as $\lambda_i = i * 0.1$ for each λ_i-cut.

The construction of the relation begins with the definition of λ_0-cut, that is, the creation of equivalence classes that group similar terms with a degree greater than $\lambda_0 = 0$. In each step, the programmer will refine the similarity relation and the equivalence classes until reaching λ_{10}-cut, at that point the similarity relation is fully defined.

[6]Deductive databases are information systems in which knowledge is stored explicitly by means of facts and implicitly through rules [8].

[7]http://www.winikoff.net/wp/


Dictionary: C:\DMail\Dictionaries\Computer_Science.dic

Topic: Soft Computing

New similarity level: 0.5

splitting class: C3
- Soft Computing
- Genetical Algorithm
- Artificial Intelligence
- Expert System
- Fuzzy Clustering
- Data Mining
- Discovery Agent

new class:
- Fuzzy Logic
- Problem Solving
- Neural Network

previous partition:
0.4-cut= {"C1, C2, C3, C4, C5"}

Figure 2.5: SiLog similarity panel [27].

Once the programmer has defined and saved the similarity relation, you can use it in the SiLog interpreter, whose main window can be seen in Figure 2.6. This window is divided in three panels, where the user can enter a Prolog program, write a query and see all (exact and approximate) answers returned by the system. During the handling of the interpreter, the construction and use of the similarity relation is carried out in a completely transparent manner to the user.

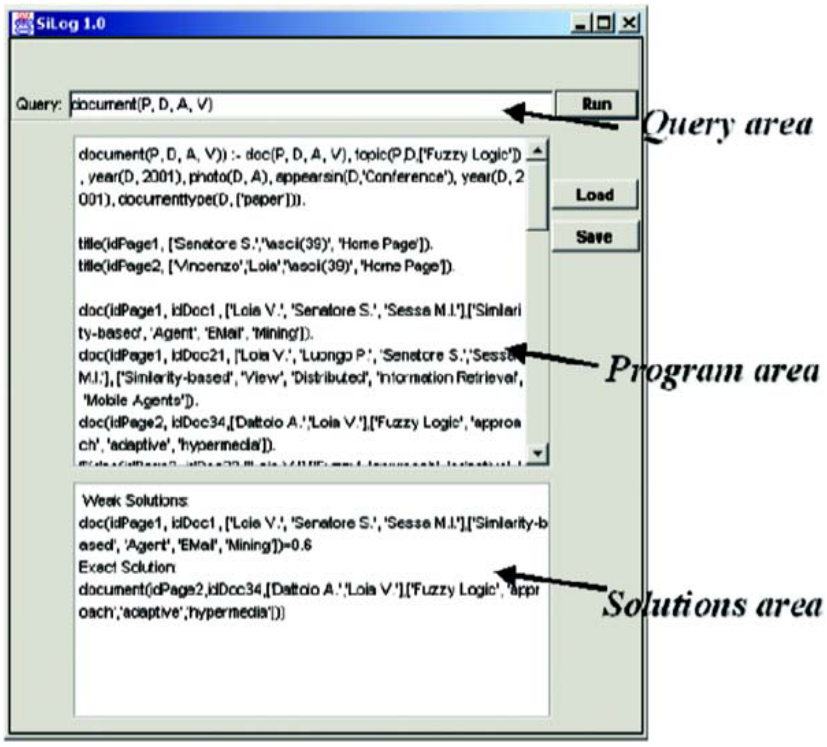
Due to its characteristics, SiLog is very suitable for the construction of deductive databases. In fact, in [27] a web architecture called Masir is exposed, related to the fields of information retrieval and knowledge discovery on the Internet, which makes use of the techniques developed in the implementation of SiLog.

2.3.3. Likelog

Likelog [9, 10, 11, 8] is a fuzzy logical programming language based on similarity relations that is distributed with a development environment called PCLikelog[8]. Both the language interpreter and the environment are implemented in Prolog.

A Likelog program consists of several files in which the program code, the similarity relation and the other resources required by the application are entered separately. In particular, each Likelog program is characterized by four files with the following content [11]:

[8]Although it is not publicly available.



```
document(P, D, A, V)
```

```
document(P, D, A, V)) :- doc(P, D, A, V), topic(P,D,['Fuzzy Logic'])
, year(D, 2001), photo(D, A), appearsin(D,'Conference'), year(D, 2
001), documenttype(D, ['paper'])).

title(idPage1, ['Senatore S.','ascii(39)', 'Home Page']).
title(idPage2, ['Vincenzo','Loia','ascii(39)', 'Home Page']).

doc(idPage1, idDoc1, ['Loia V.', 'Senatore S.', 'Sessa M.I.'],['Similari
ty-based', 'Agent', 'EMail', 'Mining']).
doc(idPage1, idDoc21, ['Loia V.', 'Luongo P.', 'Senatore S.','Sessa
M.I.'], ['Similarity-based', 'View', 'Distributed', 'Information Retrieval',
 'Mobile Agents']).
doc(idPage2, idDoc34,['Dattolo A.','Loia V.'],['Fuzzy Logic', 'approa
ch', 'adaptive', 'hypermedia']).
```

```
Weak Solutions
doc(idPage1, idDoc1 , ['Loia V.', 'Senatore S.', 'Sessa M.I.'],['Similarity-b
ased', 'Agent', 'EMail', 'Mining'])=0.6
Exact Solution
document(idPage2,idDoc34,['Dattolo A.','Loia V.'],['Fuzzy Logic', 'appr
oach','adaptive','hypermedia']))
```

Figure 2.6: Main window of SiLog [27].

- **Source code** (*.lkl). In this file, the rules and facts of the Likelog program are stored, written according to the same syntax as in any standard Prolog implementation. An important restriction imposed by Likelog is that you can not use function symbols.
- **Clause Profiles** (*.prf). This type of files are used to declare the program predicates and specify the type of each of its arguments. From the information in this file, the system internally constructs a partition of the set of constants with as many classes as types of arguments have been declared.
- **Fuzzy relation** (*.rel). In this file the equations defining the similarity relation on the constant and/or relation symbols of the first order language induced by the Likelog program must be introduced. To do this, use the predefined predicate `eq_p` as shown below:

  ```
  eq_p(<symbol_1>, <symbol_2>, <similarity_degree>).
  ```
- **Similarity Configuration** (*.swt). Finally, this configuration file is used to establish the properties of the similarity relation on the predicate symbols and each of the classes of the set of constants. For example, with this file you could associate a classical equivalence relation to the constant symbols and a fuzzy similarity relation to the predicate symbols.

It is said that the PCLikelog environment has tools that automate part of the creation of the previous files. For example, from a list of similarity equations written by the programmer, the system can automatically generate reflective, symmetric and transitive closure. But this is a point that could not be verified.

Once a Likelog program is fully defined, it can be loaded into memory for launching queries using the Likelog interpreter. This interpreter does not implement the WSLD resolution, but uses another variant of the SLD resolution strategy called *extended SLD resolution* (e-SLD) that is based on a unification algorithm in which, after each step where different syntactic symbols are unified thanks to the similarity relation, a set of symbols is generated which the authors call *cloud*.

For a goal clause and a minimum similarity degree, the Likelog interpreter calculates all the possible solutions and shows for each of them, in addition to the computed answer substitution and the similarity degree, the list of clouds with the pairs of symbols that have had to weakly unify to reach the solution.

For example, given the program $\Pi = \{p(a)\}$, the similarity relation $R(a, b) = 0.5$ and the goal $\mathcal{Q} = (\exists X)p(X)$, an answer obtained from the Likelog system would be $\langle\{X \mapsto b\}\ ;\ \{a, b\}\ ;\ \lambda = 0.5\rangle$. In this answer, the set $\{a, b\}$ would be the cloud that indicates that, if it were not for the relation that exists between the symbols a and b, that solution could not have been obtained.

The Likelog language, as in the case of SiLog, is specially designed to be used in the context of deductive databases [9]. For this reason, the authors of the language have developed a system based on Likelog called Web Information Retrieval through Likelog (WIRL) [8], with which the user can consult several databases that are set in advance.

2.3.4. Comparative

The Table 2.4 shows a comparison of the three similarity-based fuzzy logic programming systems that have been described in the previous sections: Bousi~Prolog (low-level implementation version), SiLog and Likelog.

	Fuzzy logic programming systems		
	Bousi∼Prolog (low-level) [38]	**SiLog [27]**	**Likelog [11]**
Prolog features	Partial implementation	W-Prolog Exten. Partial implementation	Partial implementation
implementation Language	Java	Java	Prolog
Operational mechanism	WSLD Resolution	WSLD Resolution	e-SLDResolution
Definition of similarity	With proximity equations	With equivalence classes	With similarity equations
Support for function symbols	Yes	Yes	No
Support for the Prolog cut operator	Yes	No	Unknown
Publicly available	Yes	No	No
Applications	Approximate Reasoning	Deductive Databases	Deductive Databases

Table 2.4: A comparison of three similarity-based fuzzy logic programming systems.

CHAPTER 3. THE BOUSI∼PROLOG LANGUAGE

Bousi∼Prolog is a fuzzy logic programming language designed to facilitate the handling of uncertainty and inaccurate information and to make the query answering process more flexible. It is the central component of the present work, from which the BPL system and its user interface have been developed.

This chapter describes in detail the characteristics and the sentences of the Bousi∼Prolog language, shows a small sample program written in that language and, finally, explains how its translation and compilation process has been designed.

Before the release of this version 2.0, a prototype of the Bousi∼Prolog language had already been presented in documents and articles such as [16, 21]. With the elaboration of this work new features have been added to the language and some constructions that were already present in its first versions have been improved.

3.1. CHARACTERISTICS OF THE BOUSI∼PROLOG LANGUAGE

The Bousi∼Prolog language is an extension of Prolog that uses the weak SLD resolution principle (WSLD) as the operational mechanism instead of the classic SLD resolution strategy.

The main goals that are to be achieved with the incorporation of a fuzzy resolution strategy to the Prolog language are to make the query answering process more flexible, allow the manipulation of fuzzy sets and facilitate the handling of vagueness through binary fuzzy relations.

In the next sections the most outstanding characteristics of the Bousi∼Prolog language are presented and the meaning of each one of the sentences and syntactic constructions that it provides.

3.1.1. Weak unification and fuzzy relations

Bousi∼Prolog is based on WSLD resolution which replaces the classical unification algorithm with a weak unification algorithm that relies on a similarity relation. Therefore, a Bousi∼Prolog program is not only composed of a set of directives, facts, and rules, but is also made up of a similarity (or, exceptionally, a proximity) relation[1].

This similarity or proximity relation is (partially) specified through the so called *proximity equations*, which are placed at the same syntactic level as the directives and clauses. The proximity equations make use of a new predefined operator, $\sim$, to represent the similarity between two symbols, and their syntax is as follows:

```
<symbol_1> ~ <symbol_2> = <approximation_degree>.
```

The Bousi∼Prolog language, using different operators, allows the programmer to specify one or more additional fuzzy relations in a program. These relations do not necessarily have to be a similarity or proximity relation and they are not taken into account during the unification process. However, these operators offer a simple way of representing arbitrary relations that can be consulted during the execution of the programs. In particular, the Bousi∼Prolog language has six fuzzy relations whose associated operators (called *relation operators*) can be seen in Table 3.1. These fuzzy relations can be classified into three groups:

- A similarity or proximity fuzzy relation that intervenes in the weak unification and WSLD resolution processes.
- Two partial order fuzzy relations for the construction of hierarchical relationships. These relations do not participate in the weak unification nor in the resolution WSLD processes.
- Three fuzzy relations whose properties are configurable by the programmer which can choice among: reflexivity, symmetry and/or transitivity. As in the previous case, they do not intervene in the unification and resolution processes.

The Bousi∼Prolog language allows the programmer to partially specify the relations that you want to define, since the properties of each relation are guaranteed thanks to the automatic computation of the reflexive, symmetric and/or transitive *closure* of the partial relations defined in the code. That is, from the equations introduced by the programmer, the Bousi∼Prolog translator will take care of adding the missing equations to each relation so that they fulfill the properties that they were asinged. Thus the programmer can simply focus on writing the most relevant equations of those fuzzy relations.

As an example, the Table 3.2 shows the real equations that a fuzzy relation would have with two initial equations, in case it was configured separately with the properties of

[1]Although the weak unification algorithm used in this implementation of the Bousi∼Prolog system requires the use of a similarity relation in order to maintain its completeness property, to increase the flexibility of the language, the use of proximity relations is also allowed.

Operators	**WSLD** [†]	**Characteristics**
`~`	Yes	Similarity or proximity relation
`~> <~`	No	Partial order relations
`~1~ ~2~ ~3~`	No	Configurable relations

[†] This column indicates whether the relations are involved in the weak unification process.

Table 3.1: Available fuzzy relations in Bousi~Prolog.

Original relation	Reflexive relation	Symmetric relation	Transitive relation
`a ~1~ b = 0.8.`	`a ~1~ b = 0.8.`	`a ~1~ b = 0.8.`	`a ~1~ b = 0.8.`
`b ~1~ c = 0.5.`	`b ~1~ c = 0.5.`	**`b ~1~ a = 0.8.`**	**`a ~1~ c = 0.5.`**
	`X ~1~ X = 1.0.`	`b ~1~ c = 0.5.`	`b ~1~ c = 0.5.`
		`c ~1~ b = 0.5.`	

Table 3.2: Proximity equations automatically generated from an example relation.

reflexivity, symmetry and transitivity.

Next, we present the syntax and meaning of the Bousi~Prolog constructions for specifying fuzzy relations.

Proximity equations and definition of fuzzy relations

`<symbol_1> <operator> <symbol_2> = <aproximation_degree>.`

Example: `green ~ blue = 0.4.`
`phsyics ~1~ maths = 0.8.`

This is the syntax of the proximity equations and, in general, of the entries of all the fuzzy relations that can appear in a Bousi~Prolog program. The sentences of this type must be outside any directive and clause of the program, but like them, they must be ended by a point.

The *`<operator>`* of an equation can be any of the relation operators shown in the Table 3.1, while the value of *`<aproximation_degree>`* must be a real number between 0 and 1 (the equations with degree 0 are ignored). On the other hand, the symbols *`<symbol_1>`* and *`<symbol_2>`* of the equations can be of two classes: identifiers and linguistic terms (these are explained later).

The *transitivity* directive

`:- transitivity(`*`<option>`*`).`

Example: `:- transitivity(yes).`

Option	Type of relation	T-Norm
`yes`	Similarity	Minimum
`no`	Proximity	N/A
`min`	Similarity	Minimum
`luka`	Similarity	Łukasiewicz
`product`	Similarity	Product

Table 3.3: Options of the *transitivity* directive.

The `transitivity` directive allows activate or deactivate the computation of the transitive closure of the fuzzy relation that intervenes in the weak unification process. In other words, this directive determines whether the main fuzzy relation associated with a Bousi∼Prolog program is going to be a similarity or proximity relation. In addition, with this same directive you can also indicate the t-norm that you want to use in the computation of the transitive closure.

The *`<option>`* parameter that the directive receives can take five different values. The meaning of all of them is explained in Table 3.3. The definition of the three t-norms supported by the language can be found on Page 9.

If this directive is not present in a program, by default it will work with a proximity relation.

The *fuzzy_rel* directive

```
:- fuzzy_rel(<operator>, [<prop_1>, <prop_2>, ...]).
```

Examples:
```
:- fuzzy_rel(~3~, [symmetric]).
:- fuzzy_rel(~>, [reflexive, transitive(product)]).
```

With this directive you can set the desired properties for the three fuzzy relations which are configurable by the programmer: ∼1∼, ∼2∼ y ∼3∼.

The first argument of the directive, *`<operator>`*, can be any of the three previous operators, while the second parameter must be a list composed of none, one or several of the following properties:

- `reflexive`, to generate a reflexive fuzzy relation.
- `symmetric`, to generate a symmetric fuzzy relation.
- `transitive`, to ensure that the relation fulfills the transitive property. Optionally, this property can include an option in parentheses that indicates the t-norm to be used

to compute the transitive closure. The options allowed are the same as those seen for the `transitivity` directive (see Table 3.3).

Exceptionally, the `fuzzy_rel` directive can also be used to choose the t-norm with which the transitive closure of the Bousi∼Prolog partial-order fuzzy relations (i.e.: ∼> and <∼) will be calculated. In this case, it is mandatory that the list of properties contains only the elements `reflexive` and `transitive` to ensure that the relation is a partial order, but freedom is left in the choice of the t-norm.

By default, unless their properties are overwritten with a directive of that type, the three configurable fuzzy relations of a program are similarity relations and the t-norm that is used in the computation of the transitive closure of all those relations, including partial order relations, is the minimum t-norm.

Unification expressions and term comparison

<term_1> <operator> <term_2> <comparator> <degree>

Examples:
```
blond ∼ brown = X
science_fiction(X) ∼2∼ mystery(X) > 0.5
```

In a query or the body of a Bousi∼Prolog program clause it is possible to use these expressions to check if two terms are related to each other, and compare the resulting approximation degree with a given value.

As in a proximity equation, the *<operator>* must be one of the Bousi∼Prolog operators shown in Table 3.1. The terms *<term_1>* and *<term_2>* can be identifiers, variables or linguistic terms as well as any type of compound term.

With regard to the *<comparator>*, this can be the unification operator, =, or any Prolog arithmetic comparison operator: `=:=`[2], `=\=`[3], `>=`, `=<`, `>` o `<`. Finally, the *<degree>* of the expression must be a real number between 0 and 1, that is the value with which the approximation degree between the two specified terms is compared.

If, instead of comparing, you want to retrieve the value of the existing relation between the terms *<term_1>* and *<term_2>*, you can pass a variable as the *<grade>* whenever the *<comparator>* is the unification operator.

When the ∼ operator is used in an expression of this type, what it is really being done is to check if two terms weakly unify according to the weak unification algorithm presented

[2]Prolog equality operator, equivalent to == in other programming languages.

[3]Prolog inequality operator, equivalent to != in other programming languages.

in Section 2.2.2. Thus, in that case the symbol ∼ is denoting a weak unification operator, the equivalent in Bousi∼Prolog to the standard Prolog unification operator, `=`.

If an operator other than ∼ is used, the weak unification algorithm is no longer applicable because the user-defined fuzzy relations (`∼1∼`, `∼2∼` and `∼3∼`) can be configured so that they are not reflexive or symmetric (two indispensable conditions for the unification algorithm to work). For this reason, when using a relation operator that is not ∼, the expression must be interpreted as an extension of that fuzzy relation to the domain of terms and the result of which is calculated according to the following formula:

$$\hat{R}(t_1,t_2) = \begin{cases} 1 & \text{if } t_1 \text{ y } t_2 \text{ are variables, } t_1 \equiv t_2 \\ & \text{and the relation } R \text{ is reflexive} \\ \alpha & \text{if } t_1 \text{ y } t_2 \text{ are constants and } R(t_1,t_2) = \alpha \\ \min\left(\alpha, \min\limits_{i=1..n}\left(\hat{R}(t_{1_i},t_{2_i})\right)\right) & \text{if } t_1 \equiv f(t_{1_1},\dots,t_{1_n}),\, t_2 \equiv g(t_{2_1},\dots,t_{2_n}) \\ & \text{and } R(f,g) = \alpha \\ 0 & \text{otherwise} \end{cases}$$

On the other hand, it is important to note that term comparison expressions (considered as a whole) are crisp and deterministic expressions, that is, either they succeed only once with approximation degree 1 or they fail (if the two terms are not related one to each other). As a clarification, the following is the behavior that a system implementing the Bousi∼Prolog language should have in the face of two scenarios:

- Given two symbols `a` and `b` which are related with degree 0.5, the expression «`a ∼ b > 0`» must succeed with degree 1 and not with degree 0.5.
- The expression «`a ∼ X =:= 1`» should only unify the variable `X` with the symbol `a` and degree 1 (even if `a` is close to several symbols with degree less than 1).

<term_1> <operator> <term_2>

Examples: `dog ∼> animal`
`small ∼1∼ X`

This comparison expression is an abbreviated form of the previous one in which the comparison operator and the final degree are omitted. It succeeds if the terms *<term_1>* and *<term_2>* are related to each other with any degree greater than 0, based on the fuzzy relation associated with the *<operator>*. Therefore, it is equivalent to:

```
<term_1> <operator> <term_2> > 0
```

The *lambda_cut* directive

`:- lambda_cut(<cut_value>).`

Example: `:- lambda_cut(0.75).`

The `lambda_cut` directive serves to establish the minimum approximation degree that will be allowed for the weak unification steps of a WSLD resolution process, and for the term comparisons made with the expressions explained above.

The *`<cut_value>`* must be a real number between 0 and 1, which is called «*lambda cut*». During a weak unification or a term comparison, all those proximity equations and any fuzzy relation whose approximation degree is lower than the *lambda cut* will be ignored.

The Table 3.4 shows, for a simple example program, what would be the result of executing two goals with different cut values (0, 0.5 and 1). As can be seen, the value 0 is the most permissive, while 1 is the most restrictive.

In a Bousi∼Prolog program the main purpose of the *lambda cut* is to limit the expansion of a WSLD resolution search tree, since its use adds one more condition to the weak unification algorithm: two symbols f and g will only be unifiable if they are related one to each other with an approximation degree greater than 0 and equal to or greater than the *lambda cut*. Formally, the use of the *lambda cut* affects rules 1 and 5 of the weak unification algorithm of Table 2.3.

The default *lambda cut* value that is applied when a directive of this type does not appear in a program, is 0.

3.1.2. Fuzzy sets and linguistic variables

The Bousi∼Prolog language allows the handling of linguistic variables according to the definition of this concept that was introduced in Section 2.1.3. Each linguistic variable is associated with a universe of discourse (in this case, numerical) and is composed of a series

Program

```
:- lambda_cut(L).
a ~ b = 0.3.
p ~1~ q = 0.8.
a.
```

Goal	L = 0	L = 0.5	L = 1
`b`	Success	Failure	Failure
`p ~1~ q`	Success	Success	Failure

Table 3.4: Application example of the *lambda_cut* directive.

of terms or linguistic labels to which it is endowed with meaning through the assignment of fuzzy subsets.

For the definition of linguistic variables the Bousi∼Prolog language provides two directives: `domain` and `fuzzy_set`. The `domain` directive allows you to specify the name and range of valid values of the universe of discourse corresponding to a linguistic variable, while the `fuzzy_set` directive is used to define fuzzy subsets associated with the terms of a linguistic variable previously declared with the `domain` directive[4].

The `fuzzy_set` directive offers the possibility of defining fuzzy subsets characterized by two types of membership functions: trapezoidal and triangular. However, the Bousi∼Prolog language also allows the use of certain linguistic values that do not need to be declared with a directive as `fuzzy_set`. These linguistic values are constructed using the operator `#` and can be of three types:

- *Domain ranges*, to represent ranges of values such as «temperature between 40 ^{o}C y 75 ^{o}C».
- *Domain points*, which allow to refer to exact values of the universe of discourse, for example «1.5 m height».
- Composite terms, which use linguistic modifiers to alter the meaning of simple terms, as in the case of «very fast».

In a Bousi∼Prolog program the information represented with the linguistic variables is transformed into a reflexive fuzzy relation, in order to facilitate the comparison between the different linguistic values of each variable. To this end, the process explained in detail in Section 2.1.4 is followed.

The fuzzy relation obtained from the linguistic variables does not constitute a new relation by itself, but rather its equations are added to the program main fuzzy relation (that participates in the weak unification process). For example, if in a Bousi∼Prolog program the linguistic variable «age» and the linguistic labels «young» and «adult» are defined, the system must internally generate a proximity equation that relates the terms «young» and «adult» with a certain approximation degree.

Below, we explain the syntax and meaning of all the directives and expressions of the Bousi∼Prolog language which are related to the handling of fuzzy sets and linguistic

[4]So, in practice the Bousi∼Prolog language focuses on the definition of the semantic component of linguistic variables, making no distinction between it and the syntactic component.

variables.

The *domain* directive

`:- domain(<name>, <minimum>, <maximum>, <measure_unit>).`

Example: `:- domain(age, 0, 100, years).`

The `domain` directive is used to declare the domain or universe of discourse of a linguistic variable called *`<name>`*, on which you can define linguistic terms.

The range of domain values is indicated by the arguments *`<minimum>`* and *`<maximum>`* , which must be both positive or negative integers. The parameter *`<measure_unit>`* represents the unit in which the elements of the domain are measured and it is purely informative.

The *fuzzy_set* directive

```
:- fuzzy_set(<domain_name>, [<label_1>(<a>, <b>, <c>, <d>),
                             <label_2>(<a>, <b>, <c>), ...]).
```

Example:
```
:- fuzzy_set(temperature, [cold(-40, -40, -10, 20),
                           warm(-10, 20, 40),
                           hot(10, 40, 80, 80)]).
```

This directive defines the fuzzy subsets associated with a series of linguistic labels â€‹â€‹belonging to the variable called *`<domain>`*, which must have been previously declared with the `domain` directive.

Each fuzzy subset is characterized by a unique name, *`<label_i>`*, and a trapezoidal or triangular membership function. The trapezoidal membership functions are defined by four integers *`<a>`*, *`<b>`*, *`<c>`* and *`<d>`*; while triangular functions only need three numbers *`<a>`*, *`<b>`* and *`<c>`*. In both cases, the numbers must be monotonically increasing and they represent the key points of the functions as can be seen in Figure 2.1.

The Bousi∼Prolog translator will transform the linguistic variables defined with the directives `domain` and `fuzzy_set` into a reflexive fuzzy relation, adding their equations to the similarity or proximity relation that takes part in the weak unification process. Therefore, all the sentences explained up to now referring to the relation $\sim$ can have as operands a linguistic term.

With the same directive `fuzzy_set` you can specify as many fuzzy subsets as you want, and several `fuzzy_set` directives can refer to the same domain.

Domain ranges

<domain_name>#<minimum>#<maximum>

Example: `height#80#120`

This term represents a linguistic label of the domain called *<domain_name>* whose membership function is a rectangular function comprised between the values *<minimum>* and *<maximum>*. In Bousi∼Prolog, these terms are called *domain ranges*.

A rectangular characteristic function only takes the value 1 for the elements of the domain located between the minimum and the maximum (both included), being 0 in the rest of cases. It can be expressed in terms of a trapezoidal function in the following way:

$$f_{rectangular}(a, b) = f_{trapezoidal}(a, a, b, b)$$

where a and b are the values of *<minimum>* and the *<maximum>* arguments, respectively.

In a Bousi∼Prolog program all the linguistic terms constructed with the operator `#`, including also those described below, are considered as identifiers, on an equal basis with other language identifiers. Therefore, they can be used as symbols of constant, function and relation or as operands of the equations of any fuzzy relation.

On the other hand, although they do not need to declare themselves with a directive `fuzzy_set`, the terms of this class are treated in the same way as the rest of the linguistic values and also take part in the generation process of the reflexive fuzzy relation (which is a component of the proximity or similarity relation used by the weak unification algorithm).

The subsets represented by the domain ranges are not fuzzy subsets, since they can only take the values 0 or 1. To define a fuzzy domain range it is necessary to add a modifier following the syntax indicated below.

<range_modifier>#<domain_name>#<minimum>#<maximum>

Example: `about#speed#100#120`

Currently, only the `about` modifier that can appear in *<range_modifier>*, which is used to "soften" the borders of a domain range. Specifically, when the `about` is applied, the following characteristic function is used [19]:

$$f_{rectangular\_about}(a, b) = f_{trapezoidal}(\max(a - \delta, n), a, b, \min(b + \delta, m))$$

where a and b are the values of the *<minimum>* and *<maximum>* argument, respectively, n and m are the minimum and maximum in the range of the domain *<domain_name>*, and δ

is a factor defined as $\delta = (m - n) * 2.5/100$.

Domain points

<domain_name>#<value>

Example: `temperature#40`

This term refers to a value of the linguistic variable called *<domain_name>* that is defined through a fuzzy point or *singleton*. Terms defined in this way are called *domain points* in Bousi∼Prolog.

A fuzzy point, as can be seen in Figure 2.1, is a fuzzy subset whose characteristic function is 1 for a certain element of the domain (in this case, the one indicated by *<value>*) and 0 otherwise.

Analogously to domain ranges, domain points can also be preceded by a modifier.

<point_modifier>#<domain_name>#<value>

Example: `about#age#36`

Currently, the Bousi∼Prolog language only allows you to use the `about` modifier in the *<point_modifier>* of a domain point expression. With this modifier, the *singleton* function of the linguistic value becomes a triangular function defined as follows [19]:

$$f_{about\_domain\_point}(x) = f_{triangular}(\max(x - \delta, n), x, \min(x + \delta, m))$$

where x is a number passed through the *<value>* argument, n and m are the minimum and maximum in the range of the domain *<domain_name>*, and δ is a factor also defined as $\delta = (m - n) * 2.5/100$.

Composite linguistic terms

<term_modifier>#<label>

Examples: `extremely#fast`
`somewhat#tall`

The fifth and last type of linguistic terms that can be constructed with the operator `#` serves to represent complex linguistic terms, formed from the concatenation of a term modifiers and a linguistic label.

In a linguistic term of this class, *<label>* must be the name of a linguistic label whose meaning is given by a fuzzy subset previously defined with a `fuzzy_set` directive, while *<term_modifier>* can be any of the four linguistic modifiers supported by the Bousi∼Prolog language: `extremely`, `very`, `more_or_less` and `somewhat`. The definition of these four modifiers can be found in Table 2.2.

3.1.3. Negation in Bousi∼Prolog

In a logic programming language such as Prolog, the negation of a goal is considered to be successful when the goal is not derivable from the program that is being executed. In other words, `not(G)` succeeds if `G` fails. This principle is known as *Negation As Failure* (NAF).

In Bousi∼Prolog, it is necessary to redefine the concept of negation to take into account the degrees of approximation of the goals. For this reason, two classes of negation are defined (both based on the NAF rule): the crisp negation, which can only succeed with degree 1; and *weak negation*, which can succeed with intermediate degrees between 0 and 1.

Crisp negation as failure

`\+(<goal>)`

Examples: `\+(white ~ black)`
`\+(young(Person) ; old(Person))`

Bousi∼Prolog uses the operator `\+` for the classical Prolog negation. A negative goal `\+(G)` fails only if the goal *<G>* succeeds with approximation degree 1. Otherwise, the negative goal `\+(G)` succeeds with approximation degree 1.

The *<goal>* of a crsip negation can be either an atomic formula or a formula formed by connectives and control predicates. It is also possible the use of specific Bousi∼Prolog sentences, such as expressions of comparison between terms.

Weak negation as failure

`not(<goal>)`

Examples: `not(hot(winter))`
`not(a ~2~ b < 0.2)`

The operator `not` represents a type of fuzzy negation that, in the Bousi∼Prolog language, is called *weak negation*. As with `\+`, a negative goal `not(G)` fails only when the argument *<G>* succeeds with approximation degree 1.

However, if the argument *<G>* is successful with degree $D < 1$, the negative goal, `not(G)`, is also successful but with an approximation of $1 - D$. In case the argument *<G>*

fails, the negation would succeed with degree 1, maintaining consistency with the classical negation.

3.2. A SIMPLE EXAMPLE

In order to see the flexibility introduced by the WSLD resolution procedure in a logic programming language like Prolog, this section will show a simple example program written in the language Bousi∼Prolog just described.

Consider again the argument that was raised in the introductory chapter of this manual:

Thrillers are interesting.
«The Treasure Island» is an adventure novel.
For me, it is more or less true that thrillers and adventure novels are similar.

Therefore, it is more or less true that «The Treasure Island» is an interesting novel.

Using the Bousi∼Prolog language, the premises of this argument could be formalized as follows:

```
interesting(Novel) :- thriller(Novel).
adventure('The Treasure Island')
thriller ~ adventure = 0.5.
```

Note that the third premise has been formalized as a proximity equation that relates the symbols `thriller` and `adventure` with a certain approximation degree. In this case, it has been considered that the expression «more or less» corresponds to a 0.5 approximation degree, although this degree would depend on the appreciation of the programmer.

Once the previous program is loaded into memory, the Bousi∼Prolog interpreter should be able to give the following answer to the query of whether «The treasure Island» is an interesting novel:

```
?- interesting('The Treasure Island').
Yes, with degree 0.5.
```

To reach this conclusion it would be necessary to take two WSLD resolution steps. In the first of them the first clause of the program would be applied, unifying the variable `Novel` with `'The Treasure Island'` to obtain the objective `thriller('The Treasure Island')` . The second step would be to unify this goal with the only fact that exists in the program, `adventure('The Treasure Island')`, which is possible thanks to the literal `thriller` and `adventure` weakly unify with degree 0.5.

If the state of a WSLD resolution is represented by a tuple consisting of the goal clause to be resolved, the substitution computed so far and the resulting approximation degree, the two steps that should be followed to reach the empty clause, in order to validate the last argument, would be the following :

$$\begin{aligned}
&\langle \leftarrow \text{interesting('The Treasure Island')}, id, 1\rangle\\
&\xrightarrow{WSLD} \langle \leftarrow \text{thriller('The Treasure Island')}, \{\text{Novel/'The Treasure Island'}\}, 1\rangle\\
&\xrightarrow{WSLD} \langle \Box, \{\text{Novela/'The Treasure Island'}\}, 0.5\rangle
\end{aligned}$$

In the Chapter 5 you can find several more examples of Bousi∼Prolog programs, in which all the characteristics of the language are used and its potential is shown.

3.3. TRANSLATION TO TPL CODE

The translation, compilation and subsequent execution of the *BPL code*, that is, of the programs written in the Bousi∼Prolog language, can be done in various ways. In Section 2.3.1, for example, the, so called, BPL *low-level implementation* was explained, in which the Bousi∼Prolog programs are translated into machine code of an extension of the abstract machine of Warren specially designed to handle fuzzy relations.

In the present work we have opted for a completely different approach, which has been called *high-level implementation*. This approach is based on building a compiler written in Prolog that takes a Bousi∼Prolog program and translates it generating an intermediate representation of the program in standard Prolog that is named *TPL code* (acronym for «BPL Translated» code).

Subsequently, with the help of a standard Prolog interpreter and auxiliary predicate modules, the TPL code can be loaded in memory like any other program written in Prolog to ask questions, which will be solved by simulating the WSLD resolution mechanism on which the Bousi∼Prolog language is based.

3.3.1. Purpose of the translation

One of the advantages offered by the Bousi∼Prolog language in the implementation of applications where fuzzy information intervenes is that it frees the programmer from having to write additional code to handle the uncertainty present in the problem we want to solve.

Following the example proposed in [29], suppose you want to define a predicate `warm_and_sunny(C)` that it is true if the city `C` has a sufficiently high average temperature («warm») and a high average number of sunshine daily hours (« sunny »). Clearly, the predicates that determine if a city is warm and sunny are fuzzy and can be satisfied with different degrees. With this in mind, the predicate could be written in standard Prolog in the following way:

```
warm_and_sunny(C, DWS) :- average_temp(C, T), warm(T, DWT1),
                          sun_hours(C, H), sunny(H, DSH2),
                          combine_degrees(DWT1, DSH2, DWS).
```

Where `DWT1` is the approximation degree to which the average temperature of `C` is considered warm; `DSH2` the approximation degree to which it can be stated that `C` is a sunny city; and `DWS` the approximation degree[5]of the answer, which is calculated by combining the degrees symbol DWT1 and symbol DSH2 (usually keeping the lesser of both).

Although the previous program is completely valid, it can be difficult to understand its expected behavior, since the logic of the problem to be solved is interleaved with the computation of the approximate degrees in the fuzzy predicates.

In the Bousi∼Prolog language, where the handling of uncertainty is integrated into the language and the approximation degrees are treated transparently to the user, the same predicate could be written like this:

```
warm_and_sunny(C) :- average_temp(C, T), warm(T),
                     sun_hours(C, H), sunny(H).
```

Considering this example with regard to the operation of the Bousi∼Prolog compiler, the last piece of code would correspond to the BPL program (the source program) that the programmer writes, while the first could be understood as the TPL code that the compiler generates automatically.[6].

Therefore, the purpose of the translating process (from BPL code to TPL code) is to

[5]Note that, in this context, an approximation degree can be thought of as a truth degree, a confidence factor or a degree of belief in the delivered answer.

[6]In this example, certain technical details have been set aside in order to facilitate understanding.

make all the modifications that are necessary to the original Bousi∼Prolog program so that, keeping its fuzzy properties, it can be executed directly by a Prolog interpreter. In other words, the compiler must generate a Prolog program whose execution under the SLD resolution mechanism be equivalent to the execution of the original Bousi∼Prolog program under the WSLD resolution strategy.

To this end, among other tasks, it will be necessary to add new arguments to the user-defined predicates (`DWS` in the example) or to make calls to predicates that do not appear in the original code (`combine_degrees` in the example).

3.3.2. Simulating WSLD resolution

As seen in the Chapter 2 (on fuzzy logic programming), the WSLD resolution strategy is essentially the same as the SLD resolution strategy, except for the use of the weak unification algorithm and the introduction of approximation degrees.

Focusing on the weak unification algorithm, the analysis performed in [16] allows us to understand this algorithm as a process consisting of two separate stages when trying to unify the atom of a goal with the head of a clause: firstly, they unify the root symbols in atom and the head and later their arguments are unified.

Therefore, in total it is possible to identify three points of the resolution process in which the WSLD and SLD resolution strategies behave differently:

a) The computation and storage of the approximation degree associated to the goals (which is a specific feature of the WSLD resolution strategy).

b) The unification of the relation symbol in the root of the selected subgoal[7] with the relation symbols rooting the heads of the program clauses.

c) The unification of the arguments of the selected subgoal with the respective arguments of the heads of the program clauses.

In order for the TPL code to correctly simulate the WSLD resolution mechanism, it is essential to modify the way in which these three phases are carried out. The techniques that have been used to adapt each one of the aforementioned phases are described below.

a) To maintain and propagate the approximation degree in which fuzzy predicates are fulfilled, we will follow proposal of [29], according to which these predicates will

[7]Remember that, in Prolog, the computation rule always selects the leftmost atom in the goal that is being solved.

include an additional output parameter where that degree to which they are satisfied will be saved.

These extra parameters will be calculated at the end of the execution of their corresponding predicates, combining all the approximation degrees that may have intervened in their resolution. Specifically, the degree resulting from the execution of a clause will be the minimum of the degrees obtained when evaluating the literals that appear in the body of the clause.

To calculate the minimum approximation degree, a call will be made to an auxiliary predicate that has been named **min_degree**. This predicate receives as a first parameter a list of approximation degrees, and returns in the second parameter the minimum of all of them.

For example, from the following rule, which defines a predicate in a Bousi∼Prologprogram:

```
p :- q, r.
```

The following predicate would be generated in the TPL code:

```
p(Degree) :-
    q(Degree1), r(Degree2),
    min_degree([Degree1, Degree2], Degree).
```

b) How to apply the weak unification algorithm to unify the selected subgoal and the heads of the program clauses is not a trivial task. Although it would be enough to execute the weak unification algorithm with both atoms (verifying if the symbols at the root are close and, in case of success, continuing with the unification of its arguments) it is advisable to delegate a part of this process to the internal resolution mechanism of the Prolog language for efficiency reasons[8].

Thus, once the closures of the fuzzy relation have been computed, in order to construct the similarity relation that will participate in the weak unification process, to force Prolog to perform the unification of the predicate symbols (i.e., the symbols rooting the subgoal and the head of the rule), each rule of the BPL program is going to be replicated in the TPL code once for each symbol that is similar to the predicate symbol of its head. This way we are able to simulate this part of the weak unification

[8]Many implementations of Prolog use *hash* tables and indexed tables to search or, at least, delimit the set of clauses that are applicable in a resolution step [15]. In this way, they unify the selected atom of a goal with the head of a clause making a very small number of checks.

process by means of the syntactic unification mechanism of the Prolog language[9]. That is, if in a Bousi∼Prolog program `p` is similar to `q` with degree 0.5 and `p` is a fact, the TPL code will have two facts: `p` and `p`. This way, when the atom `q` appears in a gaol and Prolog tries to solve it, you will find a clause in the TPL code rooted with that same symbol and Prolog will use it to continue with the resolution process. However, this step will not be able to be carried out in all cases, but it will depend on the value of the *lambda cut*. If it were set to 0.75, `p` and `q` would no longer be considered similar one to each other, so when Prolog tried to solve the atom `q` a failure should occur.

To achieve this behavior, at the beginning of all the replicated rules will execute an auxiliary predicate called **`over_lambdacut`**, which receives as a parameter an approximation degree and it is successful if and only if that parameter is equal or higher than the current value of the *lambda cut*.

On the other hand, the unification of the predicate symbols must also be taken into account in the computation of the final approximation degree of the clauses. Therefore, each replicated clause will have as an extra component in the call to `min_degree`, the approximation degree coming from the proximity equation used to replicare the original rule.

In summary, from the following BPL code fragment:

```
spring ~ summer = 0.6.
summer.
```

You would get this TPL code associated with the predicate `summer`:

```
summer(Degree) :- min_degree([], Degree).
spring(Degree) :-
    over_lambdacut(0.6), min_degree([0.6], Degree).
```

c) Finally, the weak unification of the arguments of the selected subobjective and the heads of the clauses will be done by replacing all the parameters of the heads of the clauses with variables, to later apply the weak unification algorithm to these variables and the actual arguments.

This set of unifications will be carried out by a third auxiliary predicate named **`unify_arguments`**. That predicate receives as a single parameter a list of unification problems, each problem being a list composed of three elements: the two

[9]Note at this point that we can, therefore, take advantage of the indexing mechanism of the Prolog language.

expressions to be unified and and a variable to store the resulting approximation degree. The predicate will fail when any pair of expressions is not unifiable or it is less than the current *lambda cut*.

As in the previous case, the approximation degrees resulting from each unification must be taken into consideration for the computation of the final degree associated to the clause which is being processed.

For instance, given the predicate:

```
age(manuel, adult).
```

The TPL code that is generated from it would be the following:

```
age(Arg1, Arg2, Degree) :-
    unify_arguments([[Arg1, manuel, DegreeA1],
                     [Arg2, adulto, DegreeA2]]),
    min_degree([DegreeA1, DegreeA2], Degree).
```

The combination of the three techniques explained in this section allows to translate Bousi∼Prolog programs into Prolog code that simulate the WSLD resolution strategy. The only additional requirement is that we must provide the Prolog system with the implementation of the three necessary auxiliary predicates: `min_degree`, `over_lambdacut` and `unify_arguments`.

Note, however, that in order to support all the specific sentences of the Bousi∼Prolog language, such as the comparison expressions between terms or the directives for defining linguistic variables, a translation process substantially more complex than the one presented in this section will be necessary.

CHAPTER 4. DEVELOPMENT OF THE BPL SYSTEM

The BPL system is the materialization of the Bousi∼Prolog language that has been designed in this work. It is a command line application, similar to a Prolog interpreter, through which the user can load Bousi∼Prologprograms and launch and execute Bousi∼Prologqueries.

This chapter covers the whole process of the BPL systemdevelopment, from the initial phase of requirement analysis to the final testing phase, through the analysis, design and implementation stages. Previously, in the section 4.1 it is explained which is the work method that has been followed for the elaboration of the system and the reasons of its election.

Prior to the completion of this work, the DEC-tau research group had a prototype of the BPL system [22] which partially implemented the Bousi∼Prolog language described in the Chapter 3. This prototype has served as the support to capture the requirements and the first phase of the design, but it should be noted that the BPL system presented in this project is a completely new application that implements all the characteristics of the Bousi∼Prolog language and follows a different approach both in the compilation and interpretation of the BPL code.

In particular, the original prototype of the BPL system relegated most of the parsing tasks of the BPL code to Prolog, instead of performing a real lexical, syntactic and semantic analysis as proposed in this project. On the other hand, to execute the TPL code, a meta-interpretation process was carried out, which was very expensive in terms of time and memory, since the guidelines of the section 3.3.2 were not followed to perform the translation [22].

4.1. WORK METHOD

Since Bousi∼Prolog is an extension of Prolog, from the beginning of the project it was decided to implement the BPL system in this last language, the best known and used within the logic programming paradigm.

Currently, declarative programming languages are used more frequently for research and academia than for the construction of commercial applications [15]. As a consequence, there are no specific methodologies for the development of applications in these languages, contrary to what happens, for example, with imperative or object-oriented programming languages.

However, although in the logic programming paradigm there is no commonly accepted methodology that covers the whole *software* life cycle, several techniques have been proposed that are suitable for designing and implementing programs using logical languages such as Prolog [40]. Many of them are based on a *top-down* programming style with *stepwise enhancement*, the natural way to build programs in these languages given their characteristics.

The main drawback of these techniques is that they are oriented to small scale programs, being difficult to apply to larger systems composed of several modules. In fact, the concept of module itself and the notion of modularity are not yet present in some Prolog interpreters.

Based on all the above, to develop the BPL system it has been decided to follow an incremental life cycle and to use our own development methodology composed of the five classical stages used in Software Engineering: requirement specification, analysis, design, implementation and tests.

4.2. REQUIREMENT SPECIFICATION

In this phase, a natural language description of the requirements and functionalities that the system to be built should have is made.

General requirements

It is desired to develop the so called BPL system, that allows the compilation of programs written in the Bousi$\sim$Prolog language, to launch queries to such programs and the visualization of their answers.

The system must implement the weak unification algorithm and the WSLD resolution strategy explained in Section 2.2. In addition, it has to support all the sentences and specific constructions of the Bousi$\sim$Prolog language collected in the previous chapter.

To facilitate the analysis of requirements and the subsequent stages of development, from the beginning the BPL system has been decomposed into three subsystems: a *command processor* or *command shell*, in charge of processing the user's commands; a *compiler*, whose function is to translate the code of the Bousi∼Prolog programs; and an *interpreter*[1], responsible for executing the queries and computing the answers.

Requirements for loading and compiling files

The Bousi∼Prolog compiler will be responsible for translating the BPL code files, generating TPL code (which may only contain Prolog statements) into TPL files. The compilation and translation of the BPL code should be carried out according to the process described in the section 3.3.

The TPL files generated by the compiler should be stored in the same directory in which their corresponding BPL code files are located, and will have the same name but with the extension '*.tpl*' for easy identification.

The main reason we want to store the TPL files is because we can avoid unnecessary recompilation of BPL files. When you want to load a BPL file, which has not been modified since its last compilation, the associated TPL file will be loaded without having to go through the translation process again. However, even if this is the default behavior for loading the file, if the user wants it, he must be able to force the recompilation of a BPL code file.

To facilitate the use of the Bousi∼Prologlanguage, BPL code files will be classified into two groups:

- **Programs**. Bousi∼Prolog programs are a superset of the Prolog programs and may contain any type of Prolog or Bousi∼Prolog statement, including rules, facts, directives and equations.
- **Ontologies**. Ontologies serve to define relations between terms and, as such, they can only be formed by proximity equations of fuzzy relations and directives that affect their handling or generate new equations by defining linguistic variables. An ontology can never have rules or facts.

The BPL system should allow the joint loading of a program and an ontology, so that you can use the same program with several ontologies without having to mix the program and each ontology in different code files. When the user loads a program and an ontology, the BPL system will have to combine the equations that appear in both code files to obtain the complete set of proximity equations. At any time you should be able to know what the

[1]The name "interpreter" is inherited from the times where the BPL system was a meta-interpreter. Currently, the so called "interpreter" is a module composed by a series of predicates that, essentially, give support to the implementation of the weak unification algorithm (which is the main part of the module).

program is and, if applicable, the ontology loaded in memory.

In addition to the above requirements, the system must provide a mechanism that facilitates the distribution of programs and ontologies between several files, so that the user is not forced to write all the code in a single file. Thus, although when loading a program or ontology the user will continue to enter the path of a single file, called «main file», this may contain references to other code files, called «secondary files». In any case, the compilation of all these files should continue producing a single TPL file.

Requirements for the execution of queries

The interpreter of the BPL system will be in charge of executing the *queries* launched to the system about the program loaded in memory. It will also be possible to execute queries even if no program has been loaded yet, in this case only the predefined predicates of the system can be used.

According to the translation process and execution of the BPL code that was described in section 3.3, in this project we will not create a complete/real interpreter for the Bousi∼Prolog language, rather, we want the BPL system to delegate most of the responsibility for executing the queries to the SWI-Prolog interpreter.

The queries made by the user must be able to include specific Bousi∼Prolog expressions, such as negative goals or term comparisons. In general, any expression that is valid within the body of a clause must be able to appear in a query.

When the interpreter executes a query, it must show the user if answers have been found or not for the question posed. In the affirmative case, it will have to show the approximation degree and the computed substitution answer (that is, the terms to which the free variables that appeared in the initial query have been bound). Next, in the same way as for a conventional Prolog interpreter, the user should have the possibility to request more answers.

Requirements for the command shell

The command shell is a command processor, that is, a command line application similar in structure to the *shell* of an operating system, through which the user can enter, one by one, the orders that he wants to send to the system.

The command processor must act as the public interface of BPL system and be the only one of its components that the user can see and manipulate directly. Therefore, it is essential that it has commands to access the compiler and the Bousi∼Prolog interpreter. In particular, a command will be necessary to compile and load a program or an ontology in memory; and another command to send a query to the interpreter.

To delimit the results of the queries, the value of the *lambda cut* associated to the program loaded in memory must be alowed to be modified. When the user establishes a new *lambda cut* through the command processor, this will take precedence over what could have been indicated in the program with the directive `lambda_cut` (see Section 3.1.1, page29). The user also needs to be able to see the *lambda cut* currently in use.

In order to be able to search and load files located in any folder, the command processor also has to offer the user commands to navigate through the directory structure of the computer. At a minimum, it is necessary for the user to see the name and content of the working directory and change that directory.

In addition to all the above, the command processor must implement a basic mechanism for obtaining help, through which you can consult the complete list of available commands and see the detailed description and syntax of any of them.

On the other hand, the command processor must include a series of features that make it easier for the user to work with the system, and at the same time, make it similar to other existing command interpreters with which he may have previously worked. At least the processor is required to have the following characteristics:

- Free edition of the command that is being written, which includes displacement and deletion of both characters and complete words.
- Access to a history of recent commands, containing both the commands entered in the current session and in the previous ones, up to a maximum of 100 entries.
- Intelligent autocompletion (by pressing the key *Tab*) depending on the command written so far:
  - If a command has not yet been written, the autocompletion should display a list of available commands.
  - If the load command or the change of working directory command has been typed, the autocompletion should show the names of the files and folders that are in the current working directory.
  - If a query is being written, the autocompletion has to show the names of the predicates to which it can be invoked, which includes the public predicates of

the system and those defined in the program loaded in memory.

Compatibility and support requirements

In order to Bousi∼Prolog be a useful and flexible programming language, the BPL system must be compatible with as many of Prolog predefined predicates as possible. Given that the Prolog set of predefined predicates is very broad and each implementation has its own predicate libraries, at least we have to ensure the correct functioning of the following predicates:

- Flow control (`true`, `fail`, `,`, `;`, `->`, `!`).
- List management (`member`, `append`, `length` and `reverse`).
- Analysis, construction and decomposition of terms (`functor`, `arg` and `=..`).
- Term classification (`var`, `nonvar`, `atom`, `atomic`, `compound`, `integer`, `float` and `number`).
- Integer and floating point arithmetic (`is` y al menos `>`, `<`, `=:=`, `=\=`, `+`, `-`, `*`, `/`, `mod`, `sqrt`, `floor` and `ceiling`).
- Declaration and definition of user operators (`op` and `current_op`).
- Search for all the solutions of a goal (`findall`, `bagof`, `setof`).
- Metaprogramming – dynamic modification of programs – (`dynamic`, `assert` and `retract`).
- I/O management (`open`, `close`, `read`, `write`, `nl`, `get_char` and `put_char`).
- Exception handling (`catch` and `throw`).

On the other hand, to facilitate its expansion and use, the BPL system must be able to compile and run under Windows, Linux and Mac OS X. For its implementation only the use of free tools and libraries will be allowed, not being acceptable in any case the use of not publicly available software.

4.3. ANALYSIS

This stage of development aims to organize, structure and go deeper into the requirements collected in the previous stage.

To facilitate the description of the requirements, use cases and sequence diagrams have been used. These two techniques are very common in object-oriented development method-

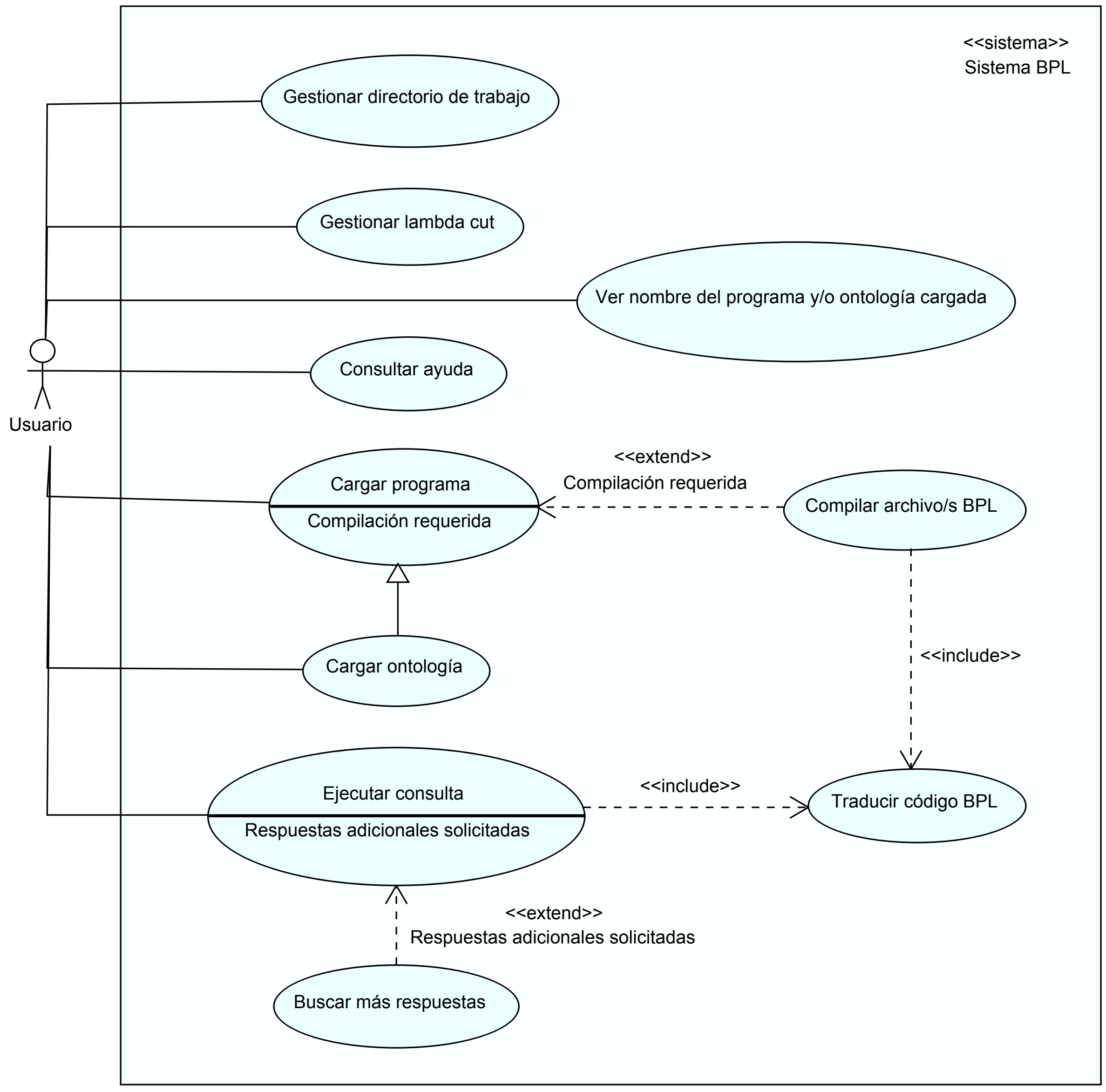


Figure 4.1: Use cases diagram of the BPL system.

ologies, but they are also applicable to this project because of their high level of abstraction and their independence from any programming language.

4.3.1. Use cases

From the requirements specification of the BPL system, ten use cases have been identified. These are represented in the use case diagram of Figure 4.1 following the standard Unified Modeling Language (UML) notation [4].

The use case diagram shows an overview of the system and the functionalities that it must implement, but it does not explain what each of them consists of. For this reason, the diagram has been complemented with a detailed specification of the six most complex use cases extracted from the requirements analysis. The behavior of the four use cases not considered, such as *Manage lambda cut* or *Consult help*, is trivial.

The specifications of the use cases indicate the actors that intervene or participate in the use case, the preconditions that must be fulfilled in order to carry it out, the postconditions or success criteria that represent the state of the system after its execution, the relationships with other use cases and a textual description of the most relevant among the main scenarios or success scenarios (error scenarios, such as when trying to load a file that does not exist or resolve a query with syntax errors, are not included).

USE CASE CU1: LOAD PROGRAM

ACTORS: User.

PRECONDITIONS: None.

POSTCONDITIONS: The program loaded in memory and the current system *lambda cut* are modified.

RELATIONSHIPS: It has an inherited use case, «load ontology» («Cargar ontología»), and an extension point that is related to the use case «Compile BPL file/s» («Compilar archivo/s BPL»).

CORRECT SCENARIO 1 (compilation not required):

1. The user enters the name of the file where the program that he wants to load is stored.

2. The system verifies that the main file of the program entered by the user exists.

3. The system verifies that the TPL file associated with the program exists.

4. The system detects that the program has *not* been modified since the TPL file was last generated, so the TPL file is updated.

5. The system loads in memory the TPL file associated with the program. If there was already a TPL file loaded in memory, it is downloaded before loading the new file.

CORRECT SCENARIO 2 (compilation required):

1. The user enters the name of the file where the program that he wants to load is stored.

2. The system verifies that the main file of the program entered by the user exists.

3. The system verifies that the TPL file associated with the program exists.
4. The system detects that the program *has been* modified since the TPL file was last generated, or the user has requested the forced recompilation of the program. This step represents the activation condition of the extension point «Required Compilation» («CompilaciÃ³n requerida»).
5. The system compiles the file/s that compose the program entered by the user to generate the corresponding TPL file. This step includes the functionality of the use case «Compile BPL file/s» («Compilar archivo/s BPL»).
6. The system checks that no serious error has occurred during the compilation, and shows the possible warnings generated in this process.
7. The system loads in memory the TPL file associated with the program. If there was already a TPL file loaded in memory, it is downloaded before loading the new file.

USE CASE CU2: LOAD ONTOLOGY

ACTORS: User.

PRECONDITIONS: There must be a program loaded in memory.

POSTCONDITIONS: The program loaded in memory and the current system *lambda cut* are modified.

RELATIONSHIPS: It is a specialization of the use case « Load program ». Inherits from it an extension point that is related to the use case «Compile file/s» («Compilar archivo/s BPL»).

CORRECT SCENARIO 1 (compilation not required):

1. The user enters the name of the ontology that he want to load.
2. The system verifies that the main file of the ontology entered by the user exists.
3. The system verifies that the main file of the program currently loaded in memory exists.
4. The system verifies that the TPL file associated with the union of the loaded program and the specified ontology exists
5. The system detects that the program and the ontology have *not* been modified since the TPL file was last generated, so it is updated.

6. The system loads in memory the TPL file associated with the union of the program and the ontology. If there was already a TPL file loaded in memory, it is downloaded before loading the new file.

CORRECT SCENARIO 2 (compilation required):

1. The user enters the name of the ontology that he want to load.
2. The system verifies that the main file of the ontology entered by the user exists.
3. The system verifies that the main file of the program currently loaded in memory exists.
4. The system verifies that the TPL file associated with the union of the loaded program and the specified ontology exists
5. The system detects that the program and/or the ontology *have been* modified since the TPL file was last generated, or the user has requested the forced recompilation of the ontology. This step represents the activation condition of the extension point «Required Compilation» («Compilación requerida»).
6. The system compiles the file/s that compose both the current program and the ontology entered by the user to generate the corresponding TPL file. This step includes the functionality of the use case «Compile BPL file/s» («Compilar archivo/s BPL»).
7. The system checks that no serious error has occurred during the compilation, and shows the possible warnings generated in this process.
8. The system loads in memory the TPL file associated with the union of the program and the ontology. If there was already a TPL file loaded in memory, it is downloaded before loading the new file.

USE CASE CU3: RUN QUERY

ACTORS: User.

PRECONDITIONS: None.

POSTCONDITIONS: None[2].

RELATIONSHIPS: It has an extension point that is related to the use case «Search more answers» («Buscar más respuestas»). Includes the use case functionality «Translate BPL code» («Traducir código BPL»).

[2]The execution of queries can cause side effects in the program loaded in memory if predicates are invoked as `assert` or `retract`, which are used to add and remove clauses respectively.

CORRECT SCENARIO 1 (one answer request for a one answer query):

1. The user enters the query that he wants to solve.
2. The system translates the BPL code of the query into TPL code. This step includes the functionality of the use case «Translate BPL code» («Traducir código BPL»).
3. The system executes the translated query on the program loaded in memory (if any) using the WSLD resolution mechanism.
4. The system notifies the user that an answer has been found, and indicates its approximation degree along with the computed substitution.

CORRECT SCENARIO 2 (two answer requests for a one answer query):

1. The user enters the query that he wants to solve.
2. The system translates the BPL code of the query into TPL code. This step includes the functionality of the use case «Translate BPL code» («Traducir código BPL»).
3. The system executes the translated query on the program loaded in memory (if any) using the WSLD resolution mechanism.
4. The system notifies the user that an answer has been found, and indicates its approximation degree along with the computed substitution.
5. The user requests a second response, that is, an additional answer. This step represents the activation condition of the extension point «Additional requested answers» («Respuestas adicionales solicitadas»).
6. The system continues the execution of the previous query. This step includes the functionality of the use case «Search more answers» («Buscar más respuestas»).
7. The system informs the user that no more answers have been found.

USE CASE CU4: SEARCH MORE ANSWERS

ACTORS: User (indirectly).

PRECONDITIONS: There must be a query running.

POSTCONDITIONS: None.

RELATIONSHIPS: Expand the functionality of the use case «Run query».

CORRECT SCENARIO 1:

1. Through the use case «Run query» («Ejecutar consulta»), the user will have requested an additional response to a query.
2. The system continues the execution of the last query.
3. The system notifies the user that an additional answer has been found, and indicates its approximation degree along with the computed substitution.
4. The user can ask again for an additional answer, in which case the functionality of this use case would be repeated.

USE CASE CU5: COMPILE BPL FILE/S

ACTORS: User (indirectly).

PRECONDITIONS: None.

POSTCONDITIONS: A TPL code file has been generated (Prolog).

RELATIONSHIPS: Expands the functionality of the use cases «Load program» («Cargar programa») and «Load ontology» («Cargar ontología») (by inheritance). It includes the functionality of the use case «Translate BPL code» («Traducir código BPL»).

CORRECT SCENARIO 1:

1. Through the use case «Load program» («Cargar programa») or «Load ontology» («Cargar ontología»), the user will have asked to load an isolated program or a program together with an ontology.
2. The system reads the source code files that compose the program and, optionally, the ontology.
3. The system performs the joint translation of the BPL source code of all read files. This step includes the functionality of the use case «Translate BPL code» («Traducir código BPL»).
4. The system generates a TPL code file with the result of the compilation.

USE CASE CU6: TRANSLATE BPL CODE

ACTORS: User (indirectly).

PRECONDITIONS: None.

POSTCONDITIONS: None.

RELATIONSHIPS: Es utilizado por los casos de uso «Compilar archivo/s BPL» y «Ejecutar consulta»It is used by the use cases «Compile BPL file/s» («Compilar archivo/s BPL») and «Execute query» «Ejecutar consulta».

CORRECT SCENARIO 1:

1. Through the use case «Compile BPL file/s» («Compilar archivo/s BPL») or «Execute query» («Ejecutar consulta»), the user will have performed an action that implies the translation of the BPL code to TPL.
2. The system translates the BPL source code of the program, the ontology or the query to TPL code (i.e., Prolog code) following, among others, the rules described in the section 3.3.2.
3. The system verifies that no serious error has occurred during the translation process (syntax error, incorrect use of a directive, etc.).

4.3.2. Sequence diagrams

To show in detail the flow of the events that occur in some use cases, and at the same time facilitate the subsequent design of the BPL system, several scenarios of the most relevant use cases of the system have been represented using the UML sequence diagrams.

Specifically, the following pages show the sequence diagrams corresponding to two success scenarios previously considered for the use cases CU1 (Load program) and CU3 (Run query).

In the sequence diagrams that have been elaborated, it is clearly indicated which part of the BPL system (command processor, compiler or interpreter) must perform each activity proposed in the use cases, which constitutes the first step in the design of the architecture of the system. From this division of tasks between the subsystems it is necessary to emphasize that the interpreter will be the one that occupies to load the programs in memory, reason for which from now on this subsystem will receive the name of «loader/interpreter».

In addition, according to the requirement specification, the diagrams clearly reflect that the command processor is the only part of the application that the user can manipulate directly.

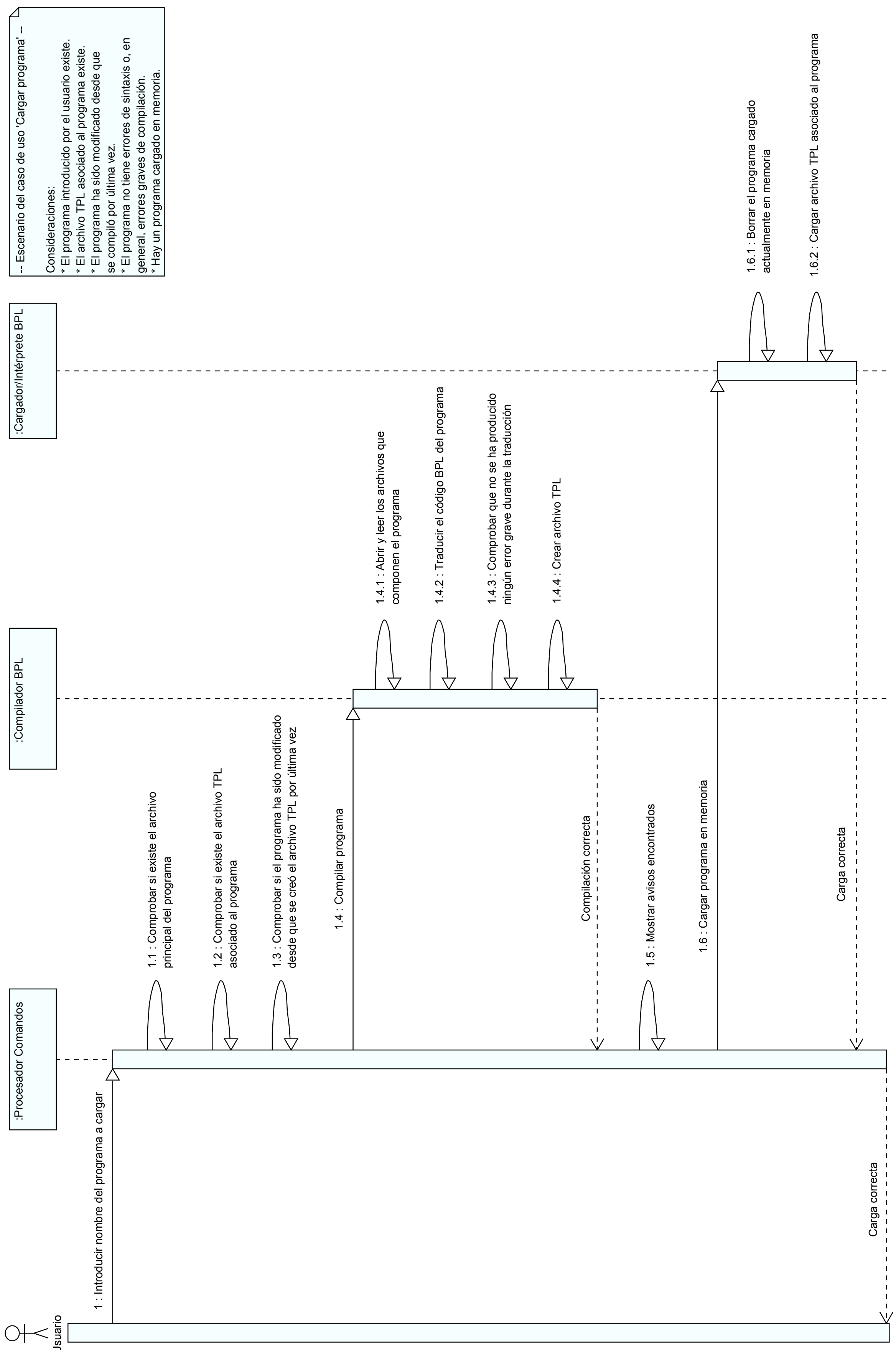

Usuario
:Procesador Comandos
:Compilador BPL
:Cargador/Intérprete BPL
-- Escenario del caso de uso 'Cargar programa' --
Consideraciones:
* El programa introducido por el usuario existe.
* El archivo TPL asociado al programa existe.
* El programa ha sido modificado desde que se compiló por última vez.
* El programa no tiene errores de sintaxis o, en general, errores graves de compilación.
* Hay un programa cargado en memoria.
1 : Introducir nombre del programa a cargar
1.1 : Comprobar si existe el archivo principal del programa
1.2 : Comprobar si existe el archivo TPL asociado al programa
1.3 : Comprobar si el programa ha sido modificado desde que se creó el archivo TPL por última vez
1.4 : Compilar programa
1.4.1 : Abrir y leer los archivos que componen el programa
1.4.2 : Traducir el código BPL del programa
1.4.3 : Comprobar que no se ha producido ningún error grave durante la traducción
1.4.4 : Crear archivo TPL
Compilación correcta
1.5 : Mostrar avisos encontrados
1.6 : Cargar programa en memoria
1.6.1 : Borrar el programa cargado actualmente en memoria
1.6.2 : Cargar archivo TPL asociado al programa
Carga correcta
Carga correcta

Figure 4.2: Sequence diagram of loading and compiling a program.

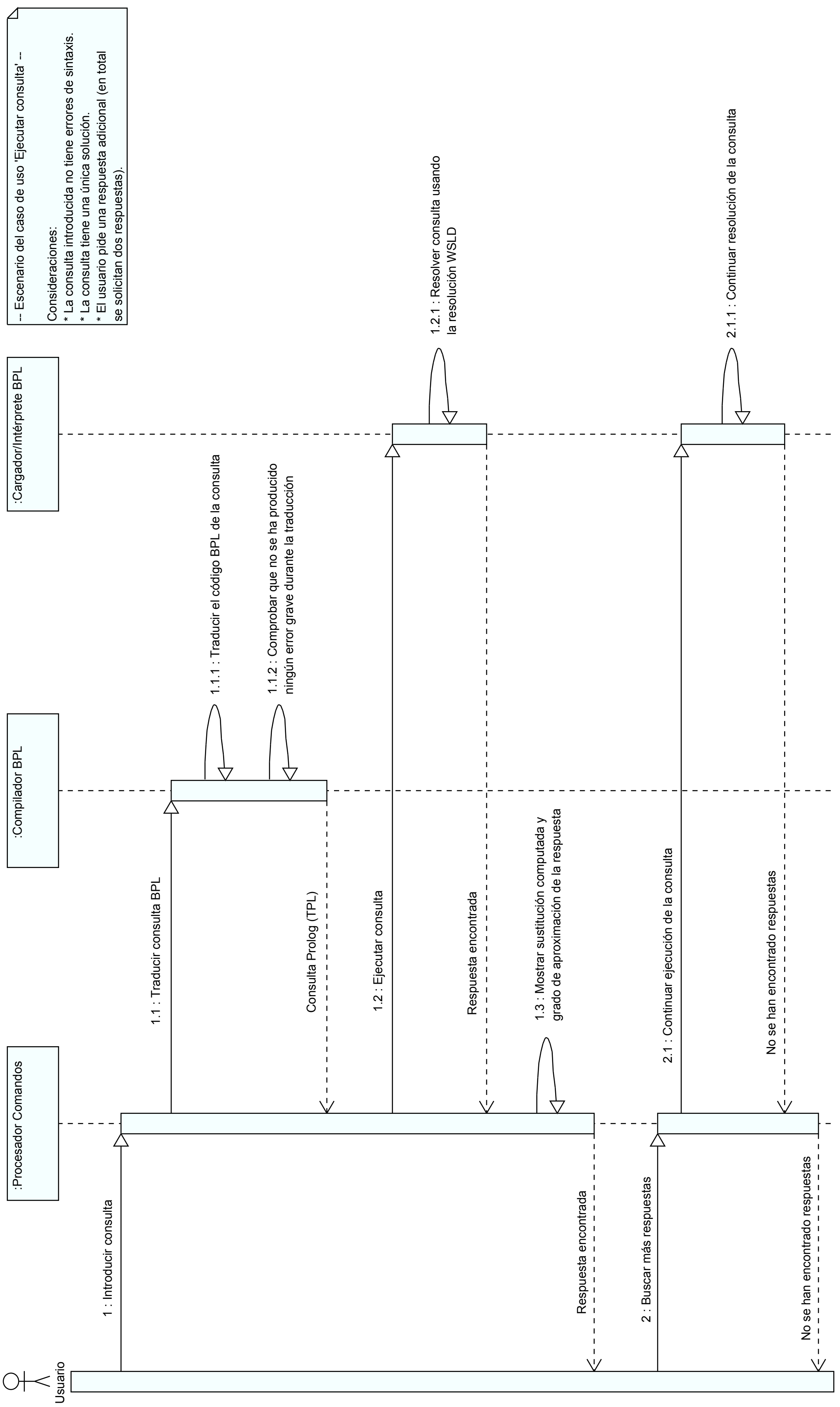

-- Escenario del caso de uso 'Ejecutar consulta' --
Consideraciones:
* La consulta introducida no tiene errores de sintaxis.
* La consulta tiene una única solución.
* El usuario pide una respuesta adicional (en total se solicitan dos respuestas).
Usuario
:Procesador Comandos
:Compilador BPL
:Cargador/Intérprete BPL
1 : Introducir consulta
1.1 : Traducir consulta BPL
1.1.1 : Traducir el código BPL de la consulta
1.1.2 : Comprobar que no se ha producido ningún error grave durante la traducción
Consulta Prolog (TPL)
1.2 : Ejecutar consulta
1.2.1 : Resolver consulta usando la resolución WSLD
Respuesta encontrada
1.3 : Mostrar sustitución computada y grado de aproximación de la respuesta
Respuesta encontrada
2 : Buscar más respuestas
2.1 : Continuar ejecución de la consulta
2.1.1 : Continuar resolución de la consulta
No se han encontrado respuestas
No se han encontrado respuestas

Figure 4.3: Sequence diagram of the execution of a query (1 answer, 2 requested).

4.4. DESIGN

In the design phase, the architecture of the system that is intended to be built is established, and both the initial structure of its components and the relationships between them are considered.

In the logic programming paradigm, which is being followed to develop the BPL system, the higher level structural components are the modules, so that to carry out the design of the system a modular approach has been followed.

On the other hand, at this stage of the development, the design method proposed by G. Karam in [24] has been used. The choice of this method is due to the fact that it has a graphic notation created specifically to show the structure of logic programs. In the Appendix C there is a description of the essential elements of the aforementioned notation.

4.4.1. System architecture

In accordance with the incremental life cycle that was adopted for the development of the BPL system, after carrying out the analysis of the requirements, the initial architecture of the system was outlined, after which it was analyzed, designed and implemented each of its modules.

The architecture of the BPL system was designed around its three main components extracted from the analysis phase: the BPL shell command processor, the compiler and the loader/interpreter. The architecture that was planned at the beginning of the design, suffered several changes throughout the project, and some modules had to be split in order to maintain a balance in the size and functionality of all of them.

Finally, the BPL system has been composed of a total of nine modules. The Figure 4.4 shows the final structure of BPL system and which are, broadly speaking, the modules that make up each of its components. The functional dependencies between the different modules of the system are represented in the diagram of the Figure 4.5.

The functionality of the modules that make up the BPL systemis briefly described below:

- ***bousi***. The module *bousi* initializes the system and launches the BPL shell command processor so that the user can start using the BPL system.
- ***bplShell***. This module implements all the functionalities related to the command

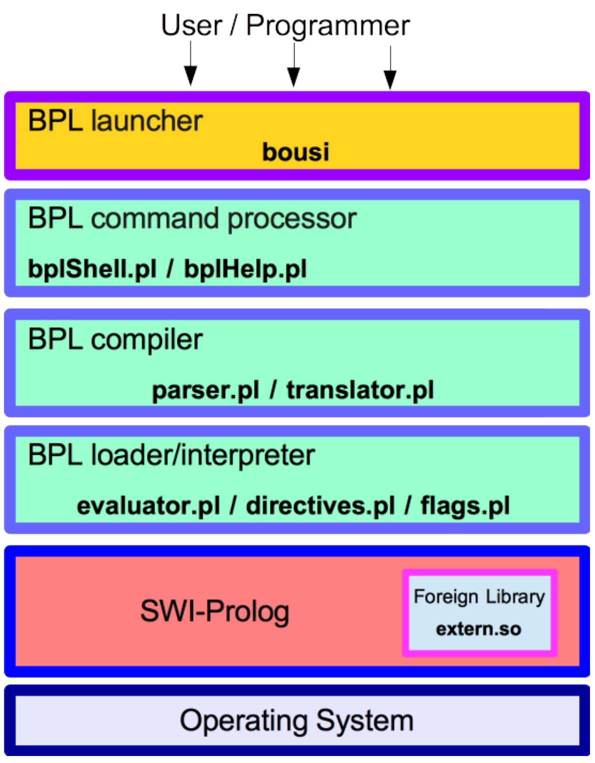


Figure 4.4: BPL system architecture.

processor, such as reading the commands entered by the user and the management of the other two subsystems (compiler and loader/interpreter).

- ***bplHelp***. This auxiliary module groups all the help messages that the system can display during its execution.
- ***parser***. The module *parser* implements the lexical, syntactic and semantic analysis of both programs and queries written in the Bousi∼Prolog language.
- ***translator***. Based on the module *parser*, this module deals with the translation of the BPL code files and the queries and the subsequent generation of the TPL code (following the guidelines of the section 3.3.2).
- ***directives***. This module is responsible for validating and executing the specific directives of the Bousi∼Prologlanguage, such as `transitivity`, `domain` or `fuzzy_set` , among others.
- ***evaluator***. The module *evaluator* implements the loader/interpreter of the BPL system. It hostes the program loaded in memory and contains the auxiliary predicates necessary to execute the TPL code of the programs with a Prolog interpreter in such a way that the WSLD resolution strategy is simulated. Among these support predicates is the weak unification predicate.

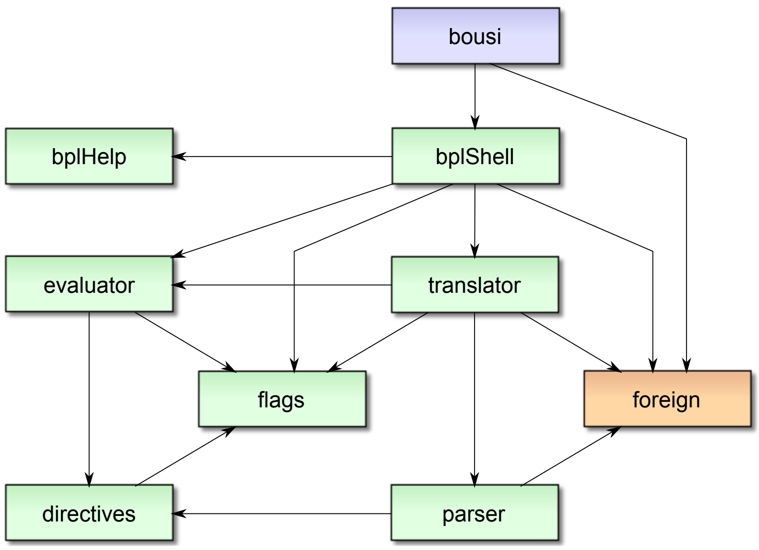


Figure 4.5: Functional dependencies of the BPL system.

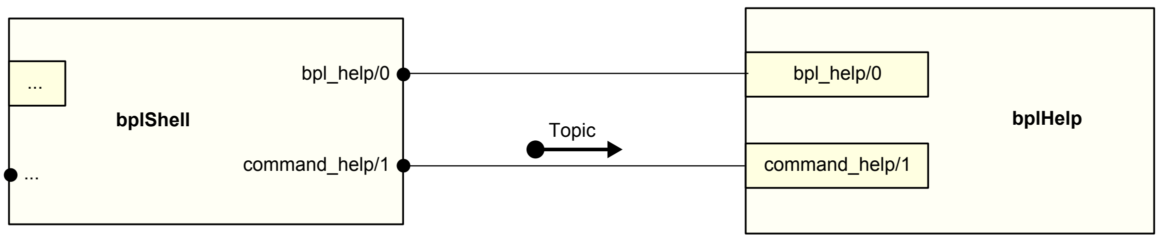
- ***flags***. This module stores and manages all those global parameters or *flags* that can be modified or consulted by the rest of the modules of the system (for example, the *lambda cut* value).
- ***foreign***. The module *foreign* is the interface to the external library of the BPL system. This library contains various predicates used by the compiler and the command processor that, for efficiency reasons, have been implemented in the C programming language. The external library contains, among others, the procedures for generating the reflexive, symmetric and transitive closures of fuzzy relations.

In addition to the previous modules, the BPL system also makes use of a module called ***utilities***, which contains various predicates for managing lists, analyzing terms, manipulating strings, etc. This module has not been included in the diagrams of the Figures 4.4 and 4.5 because it does not implement any specific requirement of the BPL system. However, it is used by most other modules to perform auxiliary tasks.

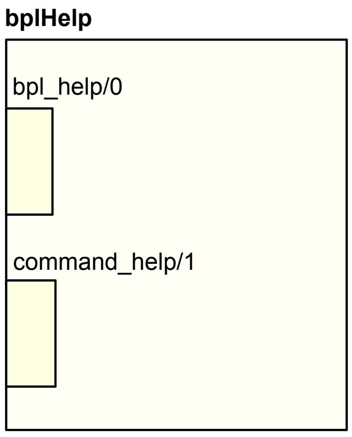
In the following sections, the functionality of each module is explained in more detail, with the help of the internal and external view design diagrams.

4.4.2. Module *bplHelp*

The module *bplHelp* shows the user all the help texts of the BPL system. As you can see in the diagram in Figure 4.6, this module has no dependencies with any other module of the system and is only used by the command processor.



(a) External view



(b) Internal view

Figure 4.6: Diagrams of the module *bplHelp*.

The two public predicates that the module modulebplHelp has are the following:

- **`bpl_help`**. Displays the complete list of commands available in the BPL system.
- **`command_help(+Topic)`**. Shows the syntax and description of the command passed as parameter.

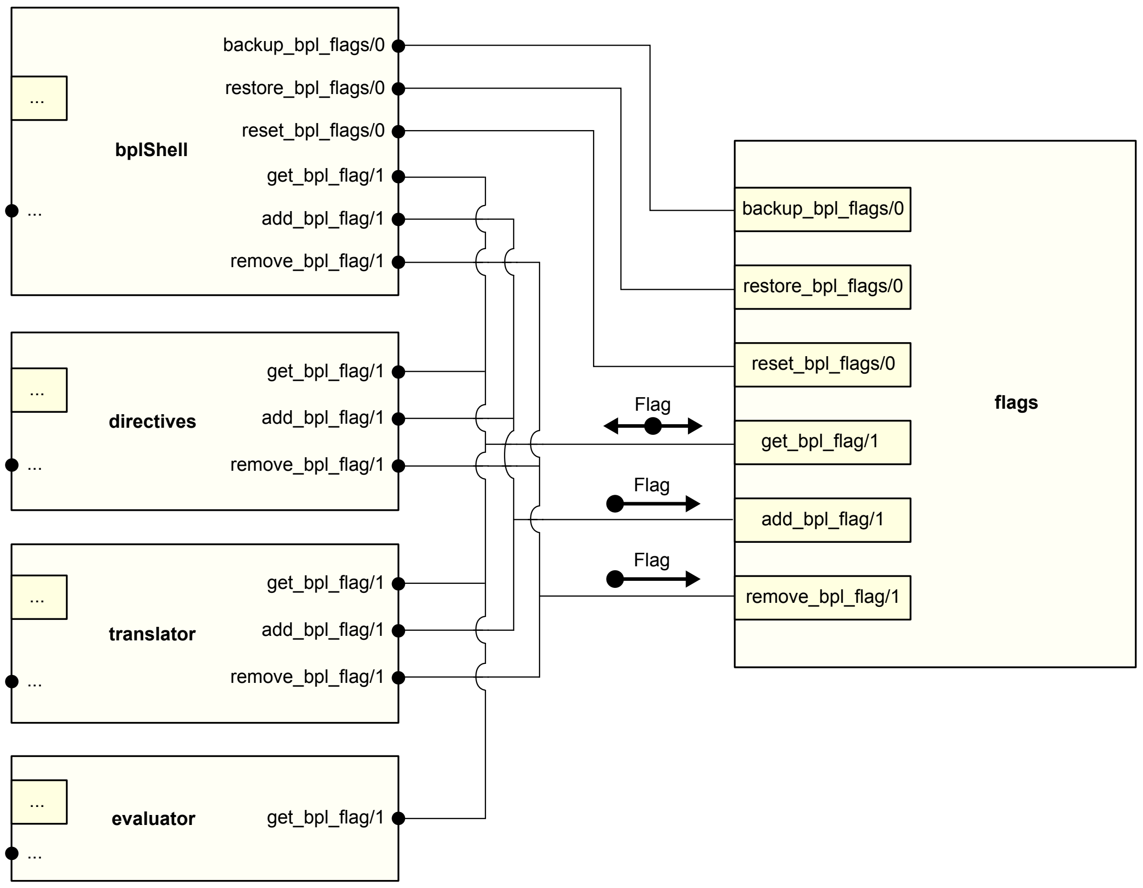
4.4.3. Module *flags*

In the BPL system there are certain configuration values and parameters that can be queried and modified by more than one module. These parameters are called *flags* and the most representative of them is the *lambda cut*, which is set when loading a program, can be modified through the command processor and is used when executing a query.

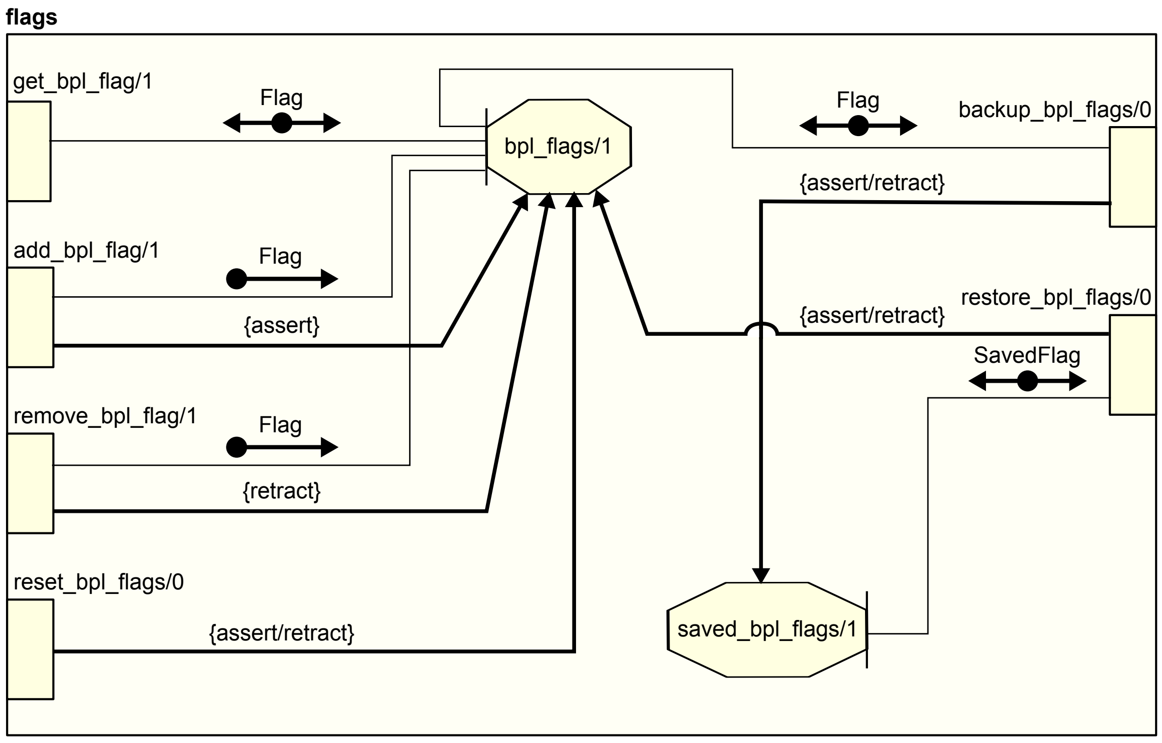
The module *flags* has been created to manage access to these global parameters in a centralized way. The module encapsulates the dynamic predicate that actually stores the *flags* and offers six public predicates to add, view, delete and restore them.

The five *flags* used by the BPL system are shown below. Most of them store the information from the specific Bousi∼Prologdirectives found in the BPL code files.

- `lambda_cut(Lambda)`. Defines the *lambda cut*, that is, the minimum approxima-



(a) External view



(b) Internal view

Figure 4.7: Diagrams of the module *flags*.

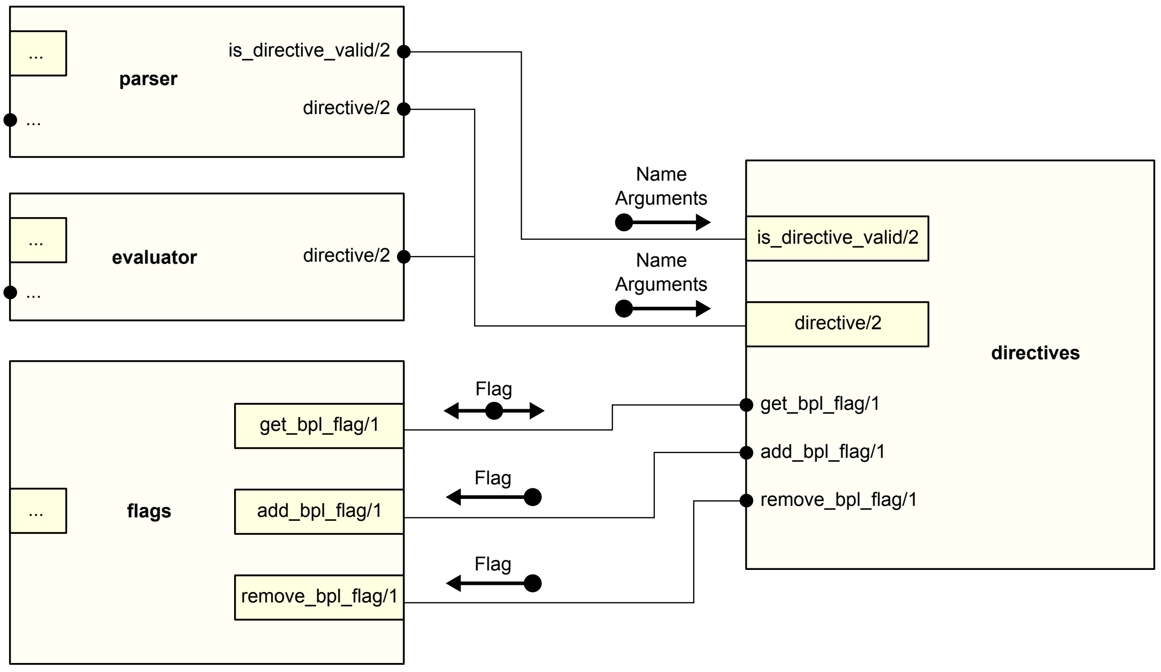
tion degree allowed for weak unifications.

- `relation_properties(Name, Properties)`. It contains the properties of the six fuzzy relations that are allowed to be defined in the Bousi∼Prolog language. `Name` indicates the name of the relation and `Properties` is a list with the same syntax as the property list of the directive `fuzzy_rel`.
- `fuzzy_domain(DomainName, Definition)`. It stores the definition of the domains or universes of discourse declared in the current program with the directive `domain` and which, as already mentioned, are associated with linguistic variables.
- `fuzzy_subsets(DomainName, Subsets)`. Contains the complete list of fuzzy subsets of each domain available in the current program. This *flag* includes both subsets defined with the `fuzzy_set` directive and subsets associated with linguistic terms built with the operator `#`.
- `program_prefix(Prefix)`. Saves the prefix associated with the program loaded in memory, which is obtained by eliminating the spaces and the extension to the program name. As will be explained later, this prefix is used to name the predicates defined by the user in the BPL code.

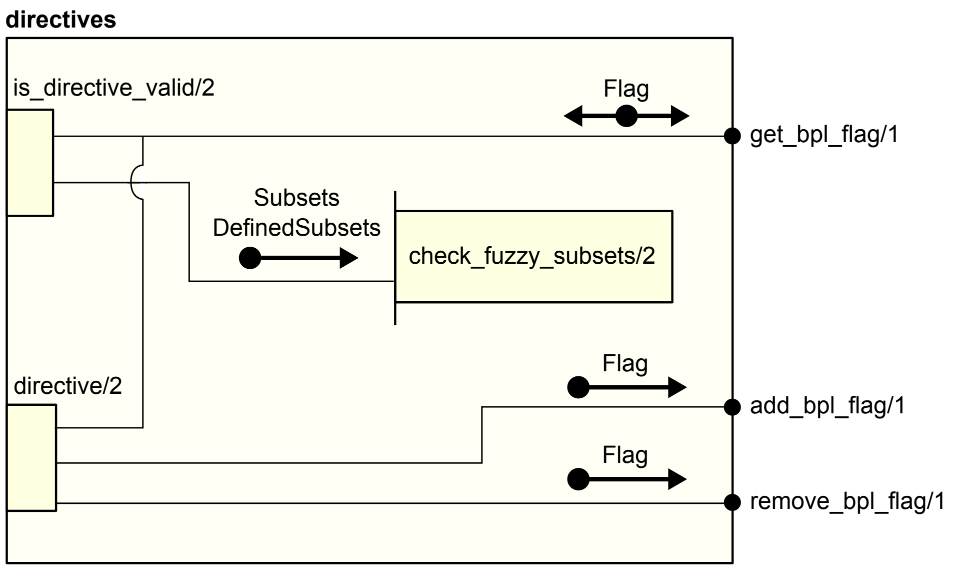
According to the diagrams in Figure 4.7 there are four modules that need access to the *flags*: *bplShell*, *directives*, *translator* and *evaluator*. The module itself does not depend on any other module in the system.

The following is a list of the six predicates that compose the public interface of this module:

- **`get_bpl_flag(?Flag)`**. Unifies the `Flag` with the current *flags* of the system. This predicate can be used both to retrieve the value of a *flag* and to check if a given *flag* is defined.
- **`add_bpl_flag(+Flag)`**. Adds the *flag* passed as a parameter to the system, first checking that it is not already defined so as not to create duplicate *flags* that may alter the consistency of the system.
- **`remove_bpl_flag(+Flag)`**. Removes the *flag* passed as a parameter from the system.
- **`reset_bpl_flags`**. Sets the default values for all the system *flags*.
- **`backup_bpl_flags`**. Stores a copy of all the current system *flags* in an auxiliary dynamic predicate.



(a) External view



(b) Internal view

Figure 4.8: Diagrams of the module *directives*.

- **`restore_bpl_flags`**. It replaces the current system *flags* with those that were saved in the last call to the **`backup_bpl_flags`** predicate.

4.4.4. Module *directives*

The module *directives* encapsulates the handling of the directives of the Bousi∼Prolog language. Its function is twofold: during the compilation of a program or an ontology, it checks the syntax and semantic aspects of the directives found in the source code files;

During the loading of a TPL file, it is responsible for executing the previously compiled directives.

All the information contained in the Bousi∼Prolog directives is moved to the system *flags* so that later it can be used by the command processor or the interpreter. For this reason, it is necessary to import the three predicates of the module *flags* that allow consulting, adding and deleting *flags*.

In the module *directives*, to simplify the validation and execution of the directives, these are represented by their name and a list with their arguments. For example, the directive `domain(age, 0, 100, years)` would be represented by the literal `domain` and the list `[age, 0, 100, years]`.

In the diagrams of Figure 4.8 can be seen that the module *directives* is used by two other modules of the BPL system: *parser* and *evaluator*. As already mentioned, its only dependency is the module *flags*.

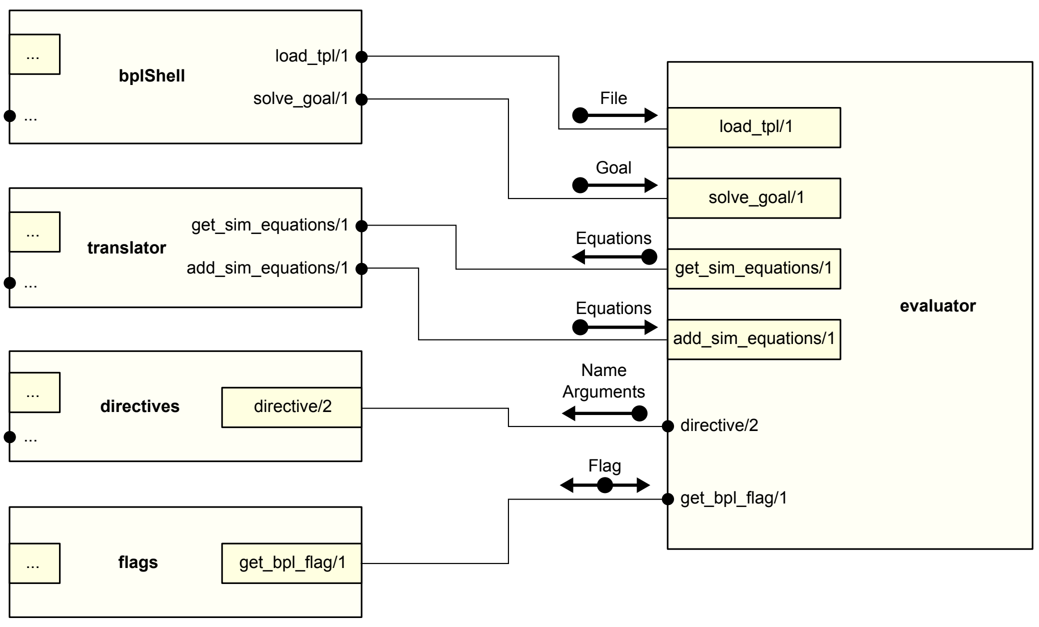
Being consistent of its double function, the module *directives* has two public predicates:

- **`is_directive_valid(+Name, +Arguments)`**. Checks that the directive name `Name` and the parameter list `Arguments` is correct. This predicate does not validate only that the arguments are of the expected type and are within the established range (for example, that the *lambda cut* is a number between 0 and 1), but it also ensures that it does not generate conflicts with previously declared directives (for example, that a domain with the same name is not defined twice).
- **`directive(+Name, +Arguments)`**. Executes the directive represented by the name `Name` and the parameter list `Arguments`. Basically, this predicate modifies the system *flags* to store in them the information contained in the directive.

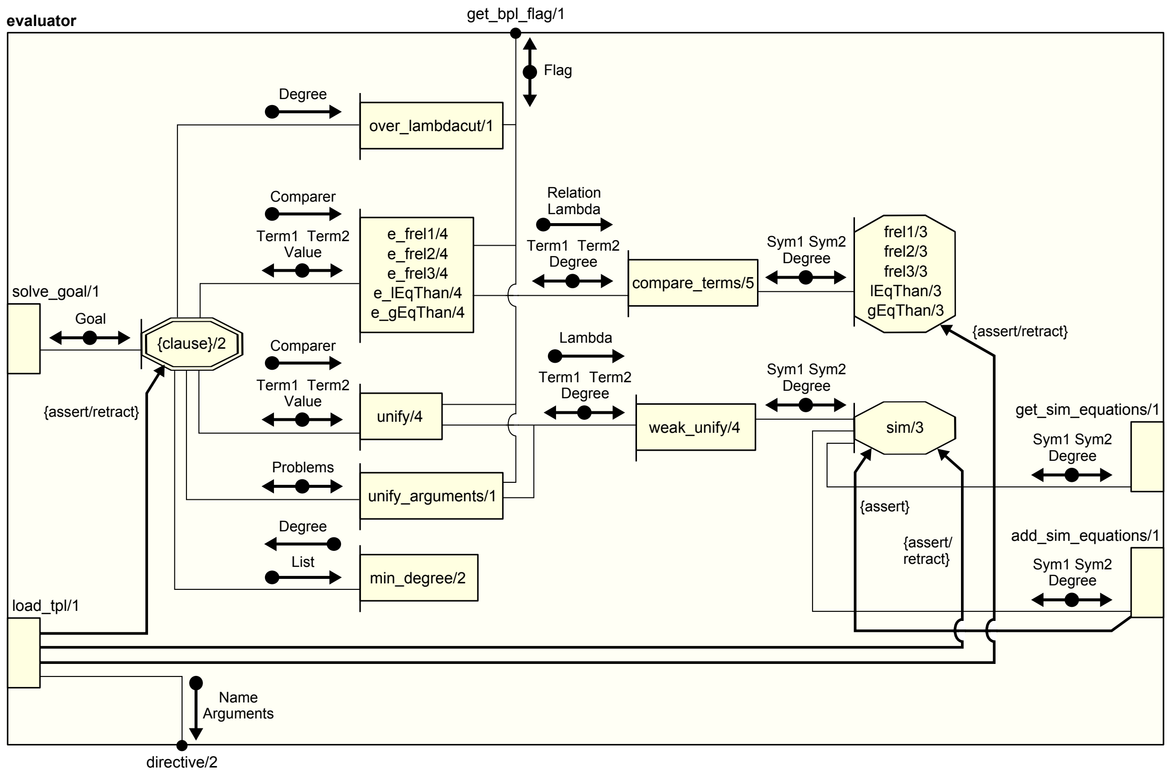
The predicate `is_directive_valid/2` makes use of the internal predicate `check_fuzzy_subsets` to check if a list of fuzzy subsets passed to the `fuzzy_set` directive is correct, based on the list of existing fuzzy subsets. This verification has been extracted from the tasks of the public predicate in order to not overload it, since the validation of the directive `fuzzy_set` is quite complex compared to the others.

4.4.5. Module *evaluator*

The module *evaluator* implements the loader/interpreter of the Bousi∼Prolog language. Therefore, its main function is to solve the queries that the user launhes through the com-



(a) External view



(b) Internal view

Figure 4.9: Diagrams of the module *evaluator*.

mand processor.

When the user loads a program or an ontology using the command processor, the TPL code file generated as a result of the compilation of the BPL file (or files) is loaded into this module. This ensures that, during the execution of the queries, the interpreter can access all the rules and proximity equations defined in the TPL file.

According to the diagrams of Figure 4.9 the module *evaluator* depends on two other modules of the system for its correct operation: *directives*, which executes the compiled directives that appear in the TPL code files; and *flags*, which is used to retrieve the current value of *lambda cut*.

The four predicates that compose the public interface of the module *evaluator* are the following:

- **`load_tpl(+File)`**. It deletes all the rules and equations of the program loaded in memory and replaces them with those containing the TPL code file passed as a parameter. If the TPL file has compiled Bousi∼Prolog directives, the module *directives* is used to execute them.
- **`solve_goal(:Goal)`**. Executes the specified query `Goal` against the program loaded in memory.
- **`get_sim_equations(-Equations)`**. Returns in `Equations` a list with all the proximity or similarity equations defined in the module *evaluator*. Each equation returned will be a term composed of the form `sim(Term1, Term2, Degree)`, for the reasons explained on the next page.
- **`add_sim_equations(+Equations)`**. Add to this module the list of proximity equations or similarity passed as a parameter. Each equation in the list must be a term composed of the form `sim(Term1, Term2, Degree).`

The clauses of the TPL file loaded in the module *evaluator* (through the public predicate **`load_tpl`**) are represented in the internal view diagram of Figure 4.9 with a dynamic predicate of arity 2 called `{clause}`[3].

Since the BPL system translates the BPL code into TPL code following the rules seen in Section 3.3.2, these dynamically loaded clauses will contain references to the three auxiliary

[3]Each clause of the TPL file loaded into memory will have as its head a different predicate identifier, which will depend on the original definition of the clause in the BPL file that gave rise to the TPL. However, in order to facilitate the reading of the diagram, all those clauses have been grouped into a single name predicate `{clause}`.

Relation	Predicate	Comparator
~	`sim`	`unify`
~1~	`frel1`	`e_frel1`
~2~	`frel2`	`e_frel2`
~3~	`frel3`	`e_frel3`
<~	`lEqThan`	`e_lEqThan`
~>	`gEqThan`	`e_gEqThan`

Table 4.1: Predicates and comparators of the fuzzy relations available in Bousi~Prolog.

predicates that were explained in the previous chapter: `over_lambdacut`, `min_degree` and `unify_arguments`. The module *evaluator* implements the aforementioned support predicates so that the TPL code can be executed properly.

On the other hand, to internally represent the equations of the fuzzy relations of BPL programs, the module *evaluator* uses six dynamic predicates, one for each relation available in the Bousi~Prolog language. All these predicates have arity 3, since each equation is defined through two symbols and their degree of approximation.

Analogously, so that during the resolution of a goal it can be verified whether two terms are unifiable or comparable according to one of the fuzzy relations of the language, the module has six other static predicates, which extend the original relations on the domain of symbols to the domain of the terms. In this case, these predicates have arity 4, since Bousi~Prolog term comparison expressions are formed by two terms, an arithmetic operator and the comparison degree (see page 27).

The names that have been chosen for the dynamic and comparison predicates of each of the six fuzzy relations of the language Bousi~Prolog are indicated in the Table 4.1. It should be noted that all these predicates are used internally by the BPL system and are not part of the Bousi~Prolog language.

Since the term comparison algorithm is identical for all fuzzy relations, in Figure 4.9 it can be seen that the predicates `e_frel1`, `e_frel2`, `e_frel3`, `e_lEqThan` and `e_gEqThan` delegate the comparison of the terms to the predicate `compare_terms`, which receives among others parameters the relations you want to use. The predicate `unify`, instead, makes a call to `weak_unify`, which implements the weak unification algorithm.

4.4.6. Module *parser*

Of the three subsystems in which the BPL system can be decomposed (command processor, compiler, and loader/interpreter), the compiler is the most complex of them because of the difficulty involved in performing a complete analysis of BPL source code files, and secondly, because of translating the BPL code into TPL code.

For this reason, although in the architecture of the system planned in the early stages of the design there was a single module called *parserTranslator* responsible for the entire compilation process, finally the compiler has been divided into two modules: *parser* and *translator*.

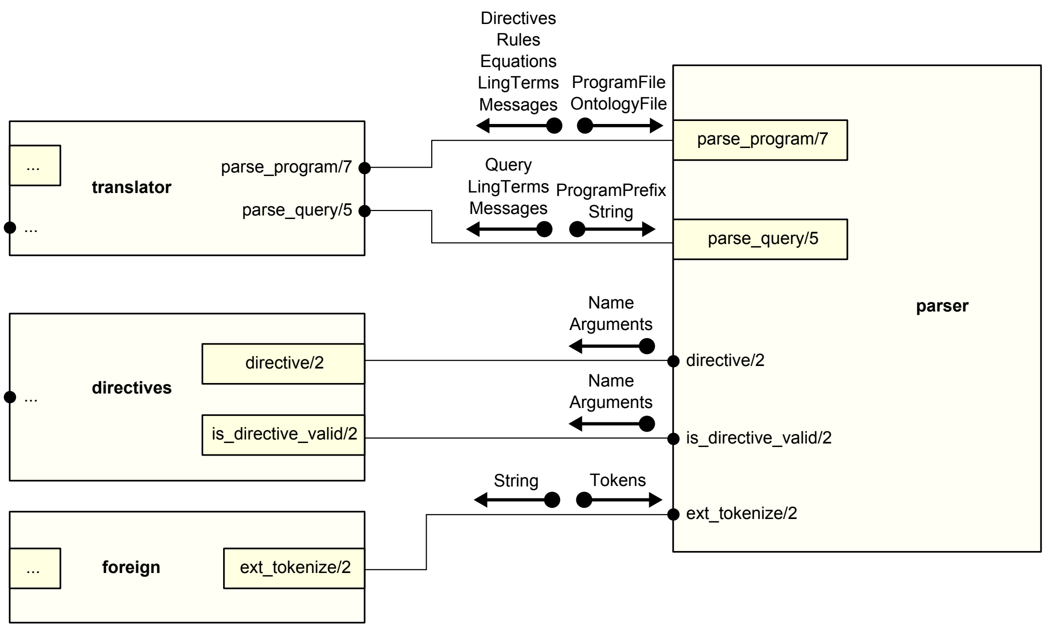
Of these two modules, the *parser* deals with the first part of the compilation process, that is, the lexical, syntactic and semantic analysis of programs, ontologies and queries written in the Bousi∼Prolog language.

Starting from a BPL source code file or a text string with a BPL query, the *parser* performs the analysis of the BPL code and generates a series of intermediate structures that can be later used by the module *translator* to build, respectively, a TPL code file or a Prolog query that can be executed on the system interpreter.

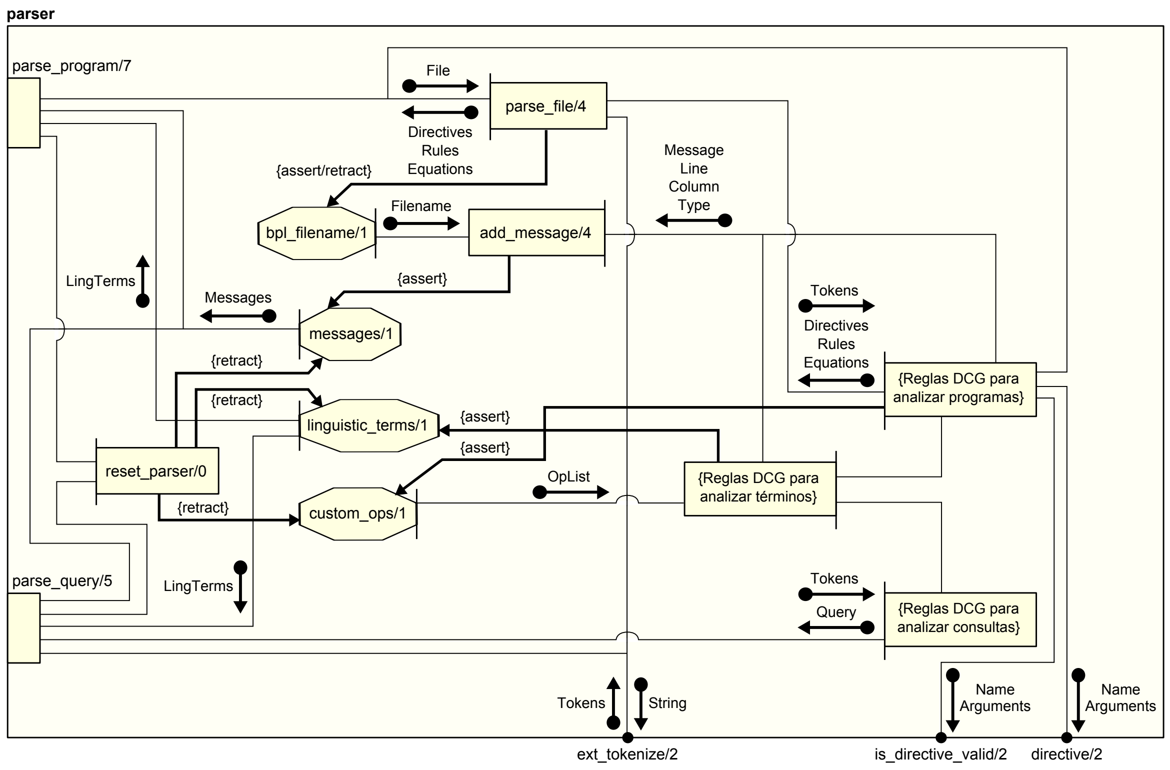
The *parser* uses the module *directives* to check the validity of the specific Bousi∼Prolog directives found in the BPL code (and execute those that are correct). It also makes use of the module *foreign* because the lexical analysis is delegated to an external predicate for efficiency reasons.

As indicated in Figure 4.10, the public interface of this module consists only of the following two predicates:

- **`parse_program(+ProgramFile, +OntologyFile, -Directives, -Rules, -Equations, -LingTerms, -Messages)`**. Performs the joint analysis of the BPL code files passed to `ProgramFile` and `OntologyFile` (which must be an ontology), and returns in each list the directives, rules, equations and linguistic terms found in them. The `OntologyFile` parameter can be omitted to load only one program. On the other hand, in the parameter `Messages` a list with the warnings and error messages generated during the compilation is returned.
- **`parse_query(+ProgramPrefix, +String, -Query, -LingTerms, -Messages)`**. It analyzes the BPL query contained in the text string `String` and returns



(a) External view



(b) Internal view

Figure 4.10: Diagrams of the module *parser*.

directives.bpl
```
:- transitivity(no).
```

equations.bpl
```
young ~ adult = 0.7.
adult ~ old = 0.4.
```

program.bpl
```
young(anne).
adult(peter).
```

with_include.bpl
```
:- include('directives.bpl').
:- include('equations.bpl').
:- include('program.bpl').
```

$\equiv$

without_include.bpl
```
:- transitivity(no).
young ~ adult = 0.7.
adult ~ old = 0.4.
young(anne).
adult(peter).
```

Code listing 4.1: Example of use of the directive *include*.

the translated query along with the linguistic terms that appear in it and the messages generated during its translation. The `ProgramPrefix` parameter indicates the prefix associated with the program loaded in memory.

Internally, the module *parser* has a predicate `parse_file` that allows you to analyze a BPL source code file. This predicate is not only used to analyze the program and the ontology passed as a parameter to **`parse_program`**, but it is also invoked recursively to analyze the secondary files referenced within the program and the ontology.

In this sense, for the user can distribute their programs and ontologies in several files, the directive `include`has been added to the language. This directive receives as parameter the name or path of a BPL code file and behaves similarly to the directive `#include` of C: a BPL program with a directive `include` is equivalent to another program with the same code but in which the directive has been replaced by the content of the referenced BPL file (see the Code listing 4.1).

Ultimately, both the predicate `parse_file` and **`parse_query`** use definite clause grammars (DCG) to perform the syntactic and semantic analysis of the BPL code from the *tokens* returned by the lexical analyzer.

The DCGs are composed of a special type of rules that facilitate the handling of *tokens* and the execution of production rules. As an example, the Code listing 4.2 shows a simple grammar that recognizes the regular expression «*a*b*c**» and returns the number of letters read. As you can see, in the DCG the operator `-->` separates the left part (the non-terminal symbol) and the right part (which may contain both *tokens* and non-terminal symbols) of each rule, the *tokens* are enclosed in square brackets and the production rules rules are delimited with braces.

```
% Grammar that recognizes the regular expression "a*b*c*"
% and returns the total number of read letters

s(N) --> a(NA), b(NB), c(NC), {N is NA + NB + NC}.

a(N) --> [a], a(N1), {N is N1 + 1}.
a(0) --> [].

b(N) --> [b], b(N1), {N is N1 + 1}.
b(0) --> [].

c(N) --> [c], c(N1), {N is N1 + 1}.
c(0) --> [].

% Examples of grammar use

example1 :- phrase(s(N), [a, b, c, c]), write(N).    % write '4'
example2 :- phrase(s(N), [a, a, c, b, c]), write(N). % Fail
example3 :- phrase(s(N), []), write(N).              % write '0'
```

Code listing 4.2: Definite clause grammar that recognizes the language *a*b*c** [40].

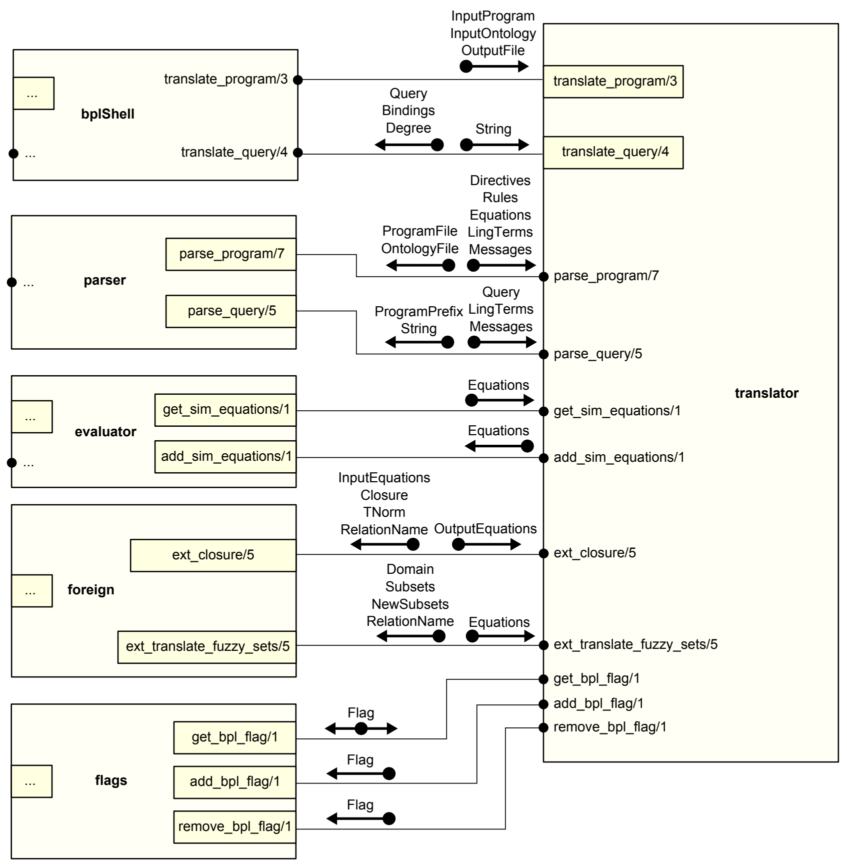
In the diagram of Figure 4.10 the rules of the DCG have been grouped to analyze the programs, the terms and the queries in three static predicates. However, when implementing grammar, as a rule you need a rule for each production of the grammar of the language you want to recognize.

4.4.7. Module *translator*

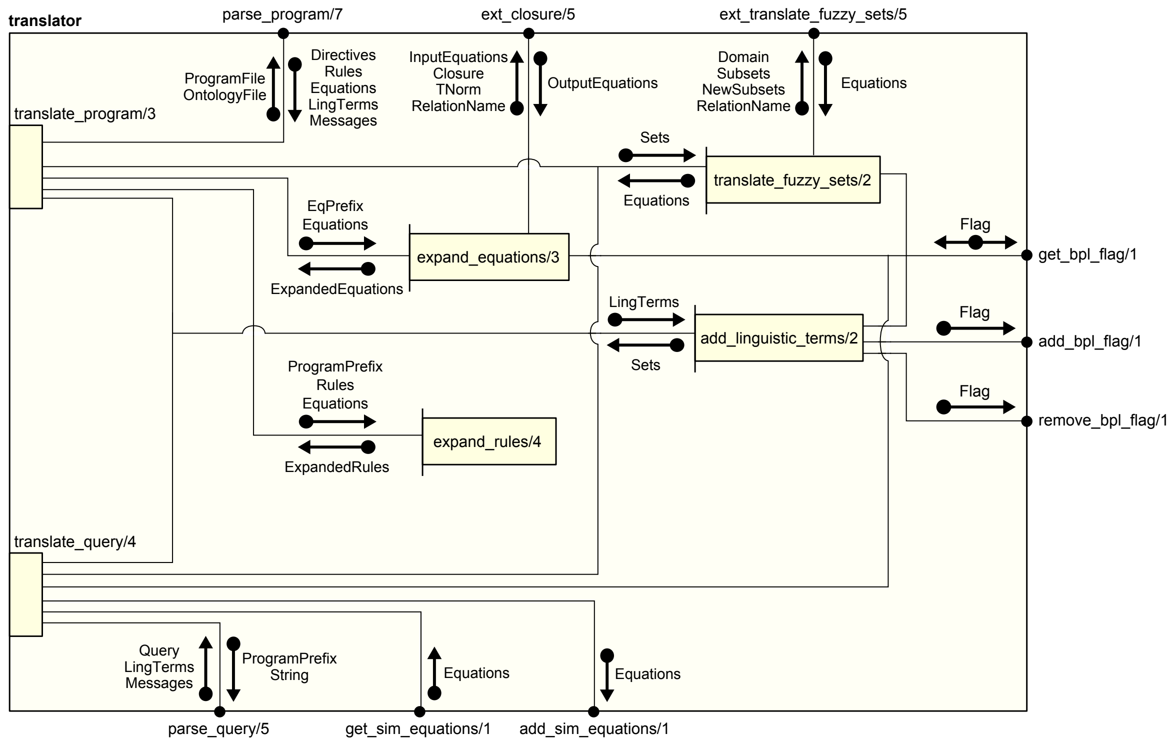
The module *translator* implements the second and last phase of the compilation process. Depending on whether you are compiling a BPL code file or translating a query entered by the user, this module is responsible for generating a TPL code file or a goal that can be sent to the interpreter of the BPL system, respectively. In both cases, the intermediate representation of the code returned by the *parser* is taken as input.

The module *translator* is only used by the command processor and depends on four other modules of the system: *parser*, *evaluator*, *foreign* and *flags*. The two public predicates that this module has are the following:

- **`translate_program(+InputProgram, +InputOntology, +OutputFile)`**. Compiles the BPL code files together `InputProgram` and `InputOntology` (which must be an ontology), and store the resulting TPL code in the file `OutputFile`. The `InputOntology` parameter can be omitted to compile only one program.
- **`translate_query(+String, -Query, -Bindings, -Degree)`**. Translates the



(a) External view



(b) Internal view

Figure 4.11: Diagrams of the module *translator*.

BPL query contained in the text string `String` and return:

- In `Query` the query TPL (Prolog) that can be executed by the BPL system interpreter.
- In `Bindings` a list of links that relates each free variable of the query to its original name. This list will be used to show the «results» of the query, that is, the computed substitution.
- In `Degree` the variable where the approximation degree of the answer will be stored, when the query be executed.

Internally, the translation of the BPL code files is divided into two clearly differentiated stages, which have been called *expansion of equations* (predicate `expand_equations`) and *expansion of rules* (predicate `expand_rules`).

The expansion of equations consists of calculating the reflexive, symmetric and/or transitive closure of the (proximity) equations introduced by the user in the BPL program, in order to generate the complete set of equations of the six fuzzy relations available in the language.

To obtain the type of closure that must be applied in each relation, the information of the directives `transitivity` and `fuzzy_rel`, which the module *directives* will have stored in the *flags* during syntactic and semantic analysis is used. To perform the computation of these closures the predicate `ext_closure` of the external library is employed.

In the Code listing 4.3 you can see an example of expansion of equations for a simple ontology formed by three equations. Notice how a similarity relation is generated from the two proximity equations (relation ~) and how the symmetric equation is added for the only equation that has the relation ~1~. In this example, in addition, you can see the translation of the Bousi~Prolog directives according to the syntax explained in the module *directives*.

On the other hand, the expansion of rules is the process by which the definitive rules of the TPL code are generated, applying the techniques seen in Section 3.3.2. From the proximity or complete similarity relation of the program being compiled, each rule of the BPL code is replicated in the TPL code once for each symbol that is close to the relation symbol that is in its head.

The Code listing 4.4 contains a fragment of TPL code generated from a program fragment composed of two proximity equations and two facts. In this fragment you can see how three rules are generated from the fact `mystery (chinatown)` because the symbol

```
% BPL Code
% ----------

:- transitivity(yes).
:- fuzzy_rel(~1~, [symmetric]).
a ~ b = 0.5.
b ~ c = 0.8.
p ~1~ q = 0.2.

% TPL Code
% ----------

:- directive(transitivity, [yes]).
:- directive(fuzzy_rel, [frel1, [symmetric]]).
sim(a, b, 0.5).
sim(a, c, 0.5).
sim(b, a, 0.5).
sim(b, c, 0.8).
sim(c, a, 0.5).
sim(c, b, 0.8).
sim(X, X, 1.0).
frel1(p, q, 0.2).
frel1(q, p, 0.2).
```

Code listing 4.3: Expansion of the equations in a BPL code fragment.

`mystery` is close to itself (because the reflexive property), to `adventure` (for the symmetric property) and to `thriller` (for the equation in the BPL code). Something similar happens with the fact `adventure(indiana_jones)`, which is "expanded" giving rise to two different rules.

In the TPL code of Code listing 4.4 it can also be seen that the identifier `prog_` has been prefixed to the heads of all program rules. This identifier represents the prefix of the program that is being compiled, and is used to distinguish two or more predicates with the same name that belong to different programs. In the example of Code listing 4.4 the file containing the source BPL program has been named *prog.bpl*.

This prefix has had to be added because the older versions of SWI-Prolog do not allow downloading a complete program once it has already been loaded into memory (except in case the same program is loaded several times, in which case the old content of the program is removed). Thus, to maintain the maximum possible compatibility, in those cases the prefix allows the predicates of a program not to conflict with those of other programs, even if they have the same predicate name.

Before performing the expansion of the equations and rules, when a program or ontology is going to be compiled in which one or more linguistic variables are defined, it is nec-

```
% BPL Code
% ----------

:- transitivity(no).
adventure ~ mystery = 0.5.
mystery ~ thriller = 0.9.
mystery(chinatown).
adventure(indiana_jones).

% TPL Code
% ----------

:- directive(transitivity, [no]).
sim(adventure, mystery, 0.5).
sim(mystery, thriller, 0.9).
sim(mystery, adventure, 0.5).
sim(thriller, mystery, 0.9).
sim(X, X, 1.0).
prog_mystery(Arg1, Degree)) :-
  unify_arguments([[Arg1, chinatown, DegreeArg1]]),
  min_degree([DegreeArg1], Degree).
prog_thriller(Arg1, Degree)) :-
  over_lambdacut(0.9),
  unify_arguments([[Arg1, chinatown, DegreeArg1]]),
  min_degree([DegreeArg1, 0.9], Degree).
prog_adventure(Arg1, Degree)) :-
  over_lambdacut(0.5),
  unify_arguments([[Arg1, chinatown, DegreeArg1]]),
  min_degree([DegreeArg1, 0.5], Degree).
prog_adventure(Arg1, Degree)) :-
  unify_arguments([[Arg1, indiana_jones, DegreeArg1]]),
  min_degree([DegreeArg1], Degree).
prog_mystery(Arg1, Degree)) :-
  over_lambdacut(0.5),
  unify_arguments([[Arg1, indiana_jones, DegreeArg1]]),
  min_degree([DegreeArg1, 0.5], Degree).
```

Code listing 4.4: Expansion of the rules in a BPL code fragment.

essary to generate the reflexive fuzzy relation associated to the fuzzy subsets of the linguistic terms. This task is occupied by the internal predicate `translate_fuzzy_sets`, which delegates the generation of the relation to the external predicate `ext_translate_fuzzy_sets`
.

Also, another important task that the module *translator* performs is the incorporation of the linguistic terms constructed with the operator `#` to the definition of the linguistic variables.

Although terms using the operator `#` do not need to be declared with the directive `fuzzy_set`, they must be taken into account for the generation of the reflexive fuzzy relation. For this reason, the module *translator* receives a list with this type of linguistic

```
% BPL Code
% ----------

:- domain(pressure, 0, 250, kpa).
:- fuzzy_set(pressure, [weak(0, 0, 30, 100),
                        normal(60, 130, 190),
                        strong(130, 190, 250)]).
current_pressure(very#weak).

% TPL Code
% ----------

:- directive(domain, [pressure, 0, 250, kpa]).
:- directive(fuzzy_set, [pressure, [weak(0,0,30,100),
                                    normal(60,130,190),
                                    strong(130,190,250)]]).
sim(very_weak, normal, 0.08).
sim(very_weak, weak, 0.888).
sim(strong, normal, 0.25).
sim(normal, very_weak, 0.08).
sim(normal, strong, 0.25).
sim(normal, weak, 0.143).
sim(weak, very_weak, 1.0).
sim(weak, normal, 0.143).
sim(X, X, 1.0).
prog_current_pressure(Arg1, Degree) :-
  unify_arguments([[Arg1, very_weak, DegreeArg1]]),
  min_degree([DegreeArg1], Degree).
```

Code listing 4.5: Translation of linguistic variables in a BPL code fragment.

terms from the *parser* and, using the add_linguistic_terms predicate, adds them to the definition of the corresponding linguistic variable stored in the *flags* before generating the equations of the relation.

The Code listing 4.5 shows a simple example BPL program that contains a linguistic variable called pressure and four linguistic terms, three defined with the directive fuzzy_set and one compound that is used in a fact (very#weak). From these terms, the nine equations[4] that appear in the TPL code are generated, in which the literal very_weak is used to represent the compound term very#weak.

In the case of translating a query, this process is slightly more complex because, if a query uses a linguistic term that has not been used until now, it is necessary to recalculate the reflexive fuzzy relation and modify the program loaded in memory[5]. The concrete steps that follow in this situation are the following:

[4]Equations whose proximity degree is 0 need not be represented.

[5]Given the large number of linguistic terms that can be defined with the operator #, it is not feasible to generate at compile time a fuzzy relation with all possible terms for each universe of discourse. That is why it is inevitable to have to recalculate and modify the relationship during the translation of the query.

a) Add the new term to the definition of its corresponding linguistic variable.

b) Calculate the relationship between the new term and all other terms of the linguistic variable to which it belongs.

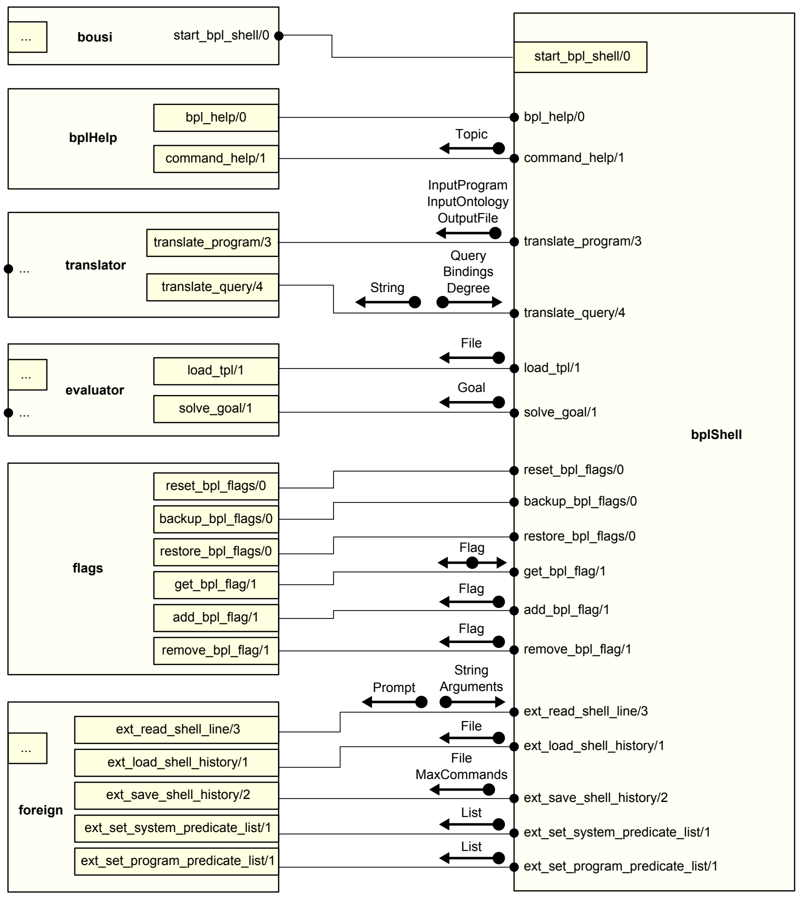
c) Add the equations obtained in the previous step to the program loaded in memory. This is the reason why the predicate `add_sim_equations` has been defined in the module *evaluator*.

4.4.8. Module *bplShell*

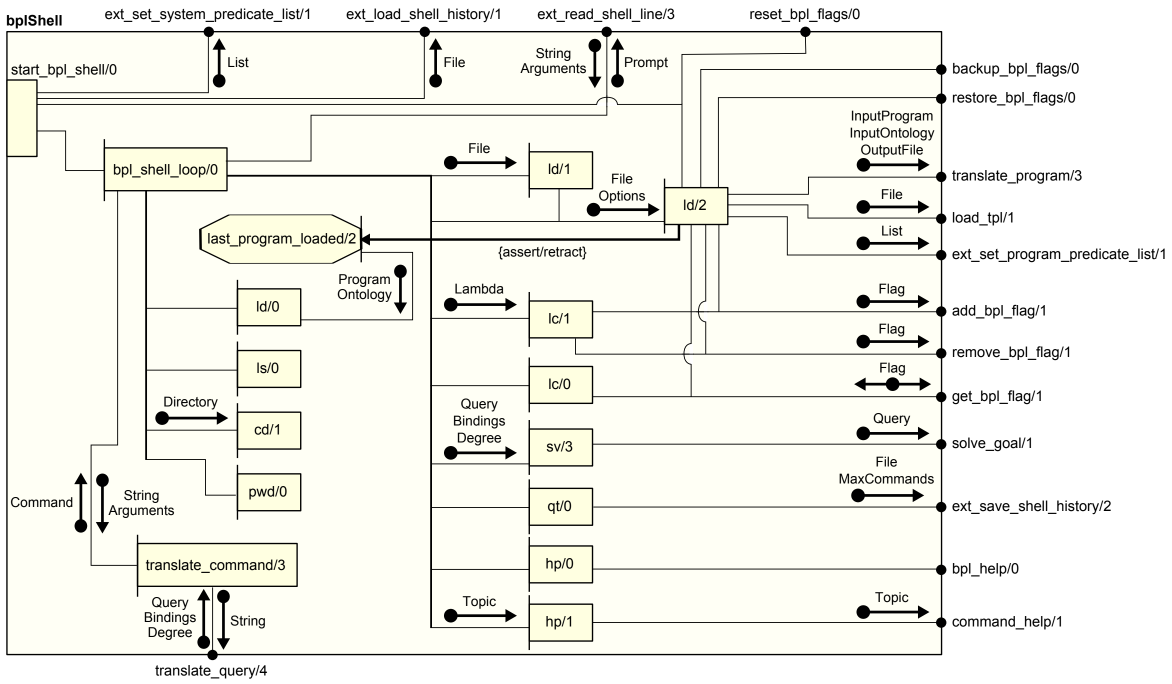
The module *bplShell* implements the command processor of the BPL system. It is the central module of the application, since it is the one that allows the user to enter the commands, receive the answers to the queries and manage the execution of both the compiler and the Bousi∼Prolog loader/interpreter.

The command processor acts as an interface between the user and the BPL system, so that it must allow the user to carry out all the functions proposed in the requirements specification and the use cases. For this task, the processor has the following 14 commands:

- `ld`: Shows the name of the program and the ontology currently loaded.
- `ld <program>`: Compiles and loads in memory the `<program>` passed as a parameter.
- `ld -f <program>`: Same as the previous command but ignores the content of the associated TPL file and *forces* the recompilation of the BPL code.
- `ld -o <ontology>`: Compiles and loads in memory the `<ontology>` passed as a parameter together with the last program that was loaded with the command `ld`.
- `ld -fo <ontology>`: Same as the previous command but ignores the content of the associated TPL file and *forces* the recompilation of the BPL code.
- `sv <query>` o `<query>`: Translates and executes the indicated `<query>`; Then, if any answer is found, it is shown to the user and the option to search for additional answers is offered. The command `sv` is the default command of the system, which means that if the user enters an order that does not conform to any of the commands shown in this list, the command will be interpreted as a Bousi∼Prolog query. In other words, the `sv` command can be ignored when writing a query.
- `lc`: Shows the current *lambda cut* value.



(a) External view



(b) Internal view

Figure 4.12: Diagrams of the module *bplShell*.

- `lc <degree>`: Set `<degree>` as the new system *lambda cut* value.
- `pwd`: Shows the path of the current working directory.
- `ls`: Displays the list of files and folders in the current working directory.
- `cd <directory>`: Sets `<directory>` as the new working directory of the system.
- `hp`: Displays the list of available commands in the shell of the BPL system.
- `hp <command>`: Shows the syntax and description of the indicated `<command>`.
- `qt`: Quits the system.

Given its status of a central module, the module *bplShell* is the one with the most dependencies of the entire system. So, you need to use predicates exported from the modules *bplHelp*, to show the help; *evaluator*, to execute queries; *translator*, to compile files and BPL queries; *flags*, to retrieve and modify some *flags*; and *foreign*, to access the support library of the command line (in order to manage the history and autocomplete functions).

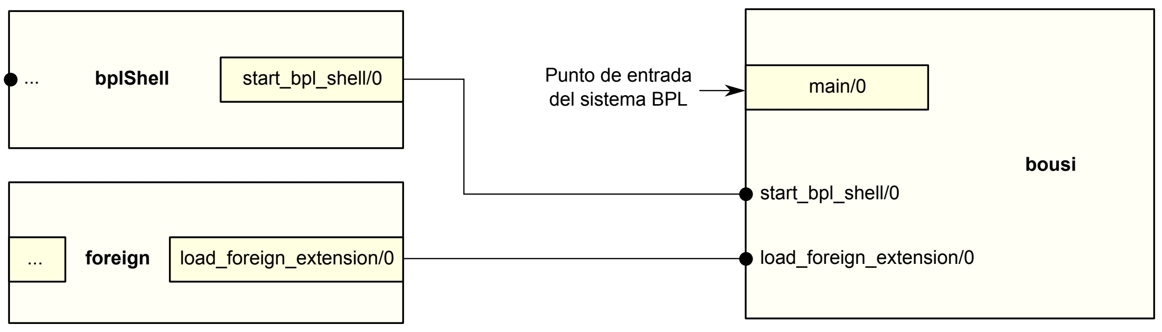
The module *bplShell* has only one public predicate, which is invoked by the module *bousi* when the system is booted:

- **`start_bpl_shell`**. Initialize and launch the command processor of the BPL system. This predicate requests an order from the user, executes it and continues to request commands from the user until the `qt` command is entered.

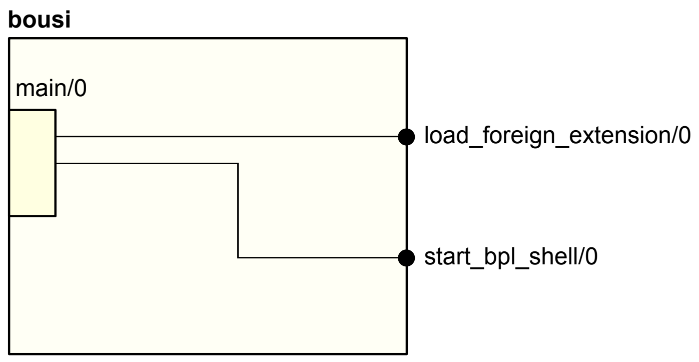
When designing this module it was decided to create an internal predicate for each order available in the command processor, as seen in Figure 4.12. Each predicate has as many arguments as parameters receives its corresponding command, with the exception of the command `ld`, where the list of options is considered as a single parameter that can receive three values: `[f]`, `[o]` or `[f, o]`.

The predicate `translate_command` is responsible for converting the text strings with the commands entered by the user in calls to any of the internal predicates of this module. When an order written by the user contains a query, the module *translator* is accessed to generate a query that can be sent to the interpreter.

In general, the execution of each of the commands is based on making calls to another module of the BPL system. The most complicated order is that of loading programs and ontologies, because the compiler and the load/interpreter must be involved in its execution. In this sense, the use of the predicates `backup_bpl_flags` and `restore_bpl_flags` should be highlighted so as not to alter the current *flags* of the system in the event of an error during the compilation of the program indicated by the user.



(a) External view



(b) Internal view

Figure 4.13: Diagrams of the module *bousi*.

4.4.9. Module *bousi*

The module *bousi* is the module that contains the entry point of the application, that is, it is the first one that is executed when the BPL systemis started.

As shown in Figure 4.13, this module has a single public predicate with no arguments called `main`. This predicate takes care of initializing the BPL system and leaving it ready so that the user can start working with it. Basically, this predicate loads the BPL foreign library, displays a welcome message to the user and then launches the command processor.

4.4.10. Module *foreign*

The module *foreign* represents the interface between the BPL system and the BPL foreign library implemented in C. This library, which connects to Prolog through the foreign interface of SWI-Prolog, contains various predicates that was implemented in C instead of in Prolog to increase the efficiency of the system.

In the BPL system there are three modules that actively use the foreign library:

- The module *parser* uses an external predicate to execute the Bousi∼Prolog lexical

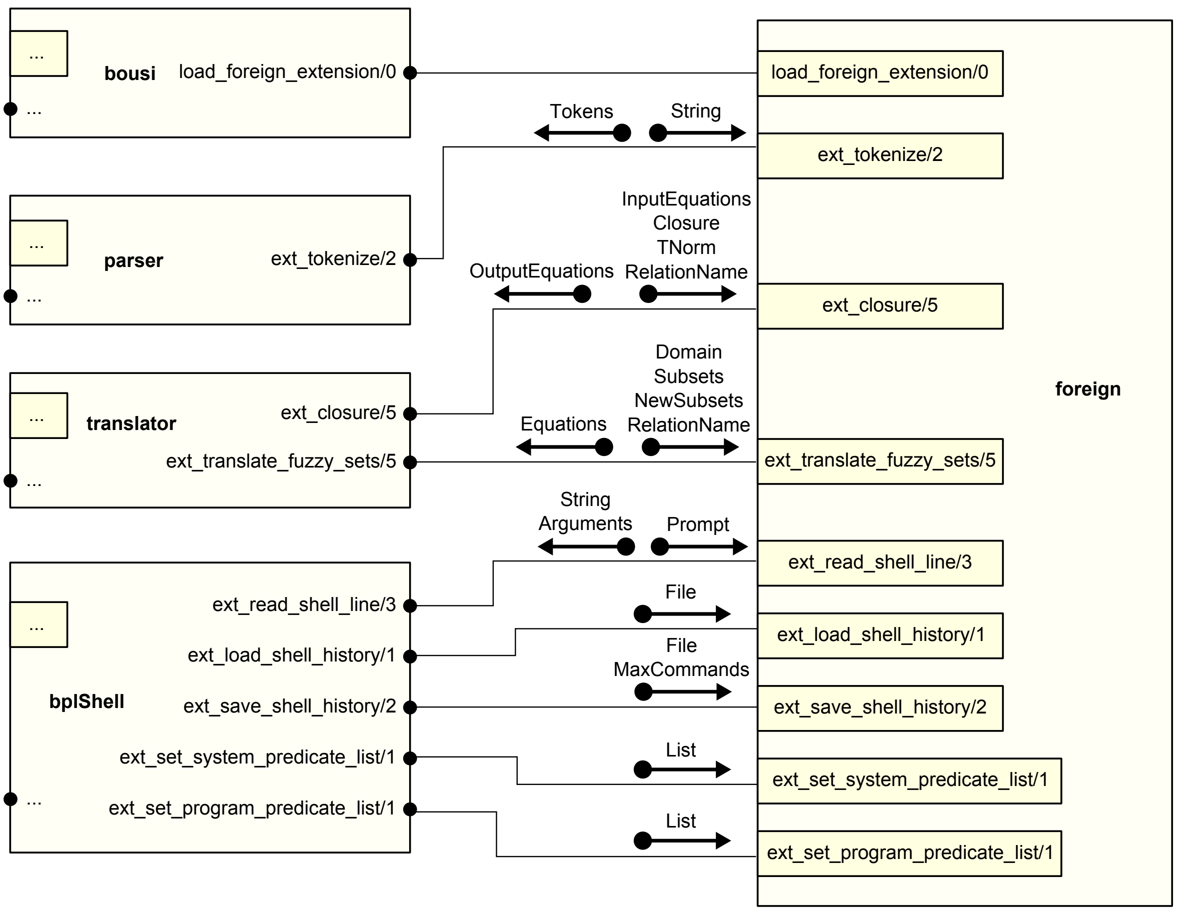


Figure 4.14: External view diagram of the module *foreign*.

analyzer, which is generated automatically with the Flex tool from the definition of the language.

- The module *translator* delegates to this library the computation of the closures of the fuzzy relations, as well as the generation of the fuzzy relation from the fuzzy subsets.
- The module *bplShell* uses the foreign library to access the functions of the support library to the command line (Editline or Linenoise, depending on the operating system).

Below is the description of each of the predicates provided by the foreign library:

- **`ext_tokenize(+String, -Tokens)`**. Lexically analyzes the string passed as a parameter, which can represent both a program and a query written in the Bousi$\sim$Prologlanguage, and returns the corresponding list of *tokens* with the format expected by the parser.
- **`ext_closure(+InputEquations, +Closure, +TNorm, +RelationName, -OutputEquations)`**. Calculates the reflexive, symmetric and/or transitive clo-

sure of the equation list `InputEquations` and returns a new list of equations in `OutputEquations` headed with the symbol `RelationName`. The parameters `Closure` and `TNorm` indicate the properties that must be applied to the relation and the type of t-norm to be used, respectively.

- **`ext_translate_fuzzy_sets(+Domain, +Subsets, +NewSubsets, +RelationName, -Equations)`**. Generates a reflexive fuzzy relation from the list of fuzzy subsets `Subsets` passed as a parameter, and returns in `Equations` the equations of the relation in which some of the subsets indicated in `NewSubsets` intervene. The argument `Domain` contains the definition of the domain to which the fuzzy subsets belong, while `RelationName` indicates the symbol that is used to head the equations.
- **`ext_read_shell_line(+Prompt, -String, -Arguments)`**. It shows to the user the indicated `Prompt` and waits for him to enter a line of text. The full line is returned in `String`, whereas the substrings into which the line is divided, using the blanks as delimiters, is returned in `Arguments`.
- **`ext_load_shell_history(+File)`**. Loads the file that contains the history of recent commands.
- **`ext_save_shell_history(+File, +MaxCommands)`**. Records the history of recent commands in the indicated file, limiting the number of commands to `MaxCommands`.
- **`ext_set_system_predicate_list(+List)`**. Sets the list of predefined predicates of the BPL system that will be used in the query autocompletion.
- **`ext_set_program_predicate_list(+List)`**. It establishes the list of predicates defined in the program currently loaded in memory, which will be accessed to automatically complete the queries.

The **`load_foreign_extension`** predicate is also part of the module *foreign* but not of the foreign library itself, since it is used when the application is started to load the library into memory.

4.5. IMPLEMENTATION

Once the system architecture has been defined and a design diagram has been drawn up for each of its components, the next step in the development is to implement these modules and connect them together to build the complete system.

In the next sections we mention the main difficulties that have been encountered during the implementation of the BPL system, and explain the functioning of some algorithms that are considered relevant (e.g., weak unification). Finally, several example sessions with the BPL system are presented, in which some of the use cases established in the analysis phase are carried out.

4.5.1. Patterns and style norms

To facilitate the understanding and reuse of the BPL system code, before starting its development, a set of rules was established in order to maintain a coherent style in the implementation of the whole system.

According to these guidelines, all the predicates of the BPL system (including those of the foreign library) are preceded by a comment in which its operation is briefly described and both the name and the type (input, output or input/output) of each of its arguments are indicated.

Thanks to the systematic writing of documentation for all the predicates, through a package included in SWI-Prolog you can automatically generate a PDF manual with the documentation of the entire system. More details on the creation of this manual can be found in Appendix A.

Finally, during the development of the BPL system some common Prolog patterns and programming techniques have been used, among which we can mention *accumulator passing*, *last call optimization* and *failure-driven loops* [40, 32].

4.5.2. Weak unification algorithm

The Bousi∼Prolog language is based on the WSLD resolution strategy which is implicitly implemented in the TPL code generated by the compiler of the BPL system, as it was explained at the end of the previous chapter.

In order for the TPL code to work correctly, it is necessary to provide the implementa-

```
% weak_unify(?Term1, ?Term2, +Lambda, ?Degree)

weak_unify(Atomic1, Atomic2, Lambda, Degree) :-
  % Atom (constant) unification
  atomic(Atomic1), atomic(Atomic2), !,
  sim(Atomic1, Atomic2, Degree),
  Degree >= Lambda.

weak_unify(Term1, Term2, Lambda, Degree) :-
  % Term decomposition
  compound(Term1), compound(Term2), !,
  Term1 =.. [Functor1|Args1],
  Term2 =.. [Functor2|Args2],
  length(Args1, Arity),
  length(Args2, Arity),
  sim(Functor1, Functor2, DegreeFunctor),
  DegreeFunctor >= Lambda,
  weak_unify_args(Args1, Args2, Lambda, DegreeArgs),
  Degree is min(DegreeFunctor, DegreeArgs).

weak_unify(Term, Variable, _Lambda, 1) :-
  % Term/variable swap +  Variable elimination
  nonvar(Term), var(Variable), !,
  Variable = Term.

weak_unify(Variable, Term, _Lambda, 1) :-
  % Variable elimination + Trivial equation removal
  var(Variable),
  Variable = Term.


% weak_unify_args(?Args1, ?Args2, +Lambda, ?Degree)

weak_unify_args([], [], _Lambda, 1).

weak_unify_args([Arg1|MoreArgs1],[Arg2|MoreArgs2],Lambda,Degree) :-
  weak_unify(Arg1, Arg2, Lambda, DegreeArg),
  weak_unify_args(MoreArgs1, MoreArgs2, Lambda, DegreeMoreArgs),
  Degree is min(DegreeArg, DegreeMoreArgs).
```

Code listing 4.6: Implementation of the weak unification algorithm.

tion, among others, of the predicate `unify_arguments`, which is used to weakly unify the real and formal parameters of the predicates.

In the BPL system, the predicate that implements the weak unification algorithm is called `weak_unify` and is a component of the module *evaluator*, since it is in this module where the dynamic predicate `sim` that contains the proximity equations of the program loaded in memory. The complete code of `weak_unify` can be found in the Code listing 4.6.

The predicate definition `weak_unify` is largely based on the rules of the weak unification algorithm of Table 2.3 that was explained in more detail in the chapter on fuzzy logic programming. The correspondence between the rules of the original algorithm and its implementation is briefly discussed below.

- **Term decomposition** This rule has been implemented by means of two clauses, to differentiate the case where the entry terms are simple (constant) or compound (i.e., general terms or atomic formulas). In the latter case, after verifying that the relation or functor symbols of both terms are close, an auxiliary predicate called `weak_unify_args` is used to weakly unify its arguments.
- **Swap** and **variable elimination** This rule partially corresponds to the third clause of the predicate `weak_unify`. In this clause, the swap and variable elimination rules of the weak unification algorithm are applied simultaneously to optimize its execution.
- **Variable elimination** and **removal of trivial equations**. These two rules of the weak unification algorithm have been merged into a single clause that acts as follows: whenever the first input term of the algorithm is a variable, it is unified with the second term, regardless of whether it is an atomic or compound term, a variable or a reference to the first term.
- **Failure rule** The rule responsible for detecting that two expressions are not unifiable has not been actually implemented. Instead, following the Prolog programming style, it is considered that two expressions are not unifiable when the predicate `weak_unify` fails.
- **Occur check** This rule has been omitted for reasons of efficiency as it is done in most Prolog systems.

It noteworthy that the composition of substitutions and the generation of the final substitution are tasks that Prolog performs automatically when an explicit unification statement is executed as `Variable = Term`.

4.5.3. Term comparison algorithm

In addition to the weak unification algorithm, the BPL system also implements the term comparison algorithm, which calculates the approximation degree between any two terms according to the formula presented in Section 3.1.1, page 28. As commented in that section, this algorithm provides the extension to the domain of terms of the respective fuzzy relations defined by the user.

Given its similarity to the weak unification algorithm, the term comparison algorithm

```
% compare_terms(+Relation, ?Term1, ?Term2, +Lambda, ?Degree)

compare_terms(Relation, Term1, Term2, _Lambda, Degree) :-
  % Variable-Variable comparison
  var(Term1), var(Term2), !,
  Term1 == Term2,
  apply(Relation, [Term1, Term2, Degree]),
  var(Term1), var(Term2).

compare_terms(Relation, Term1, Term2, Lambda, Degree) :-
  % Atomic-Atomic comparison
  atomic(Term1), atomic(Term2), !,
  apply(Relation, [Term1, Term2, Degree]),
  Degree >= Lambda.

compare_terms(Relation, Term1, Term2, Lambda, Degree) :-
  % Compound-Compound comparison
  compound(Term1), compound(Term2), !,
  Term1 =.. [Functor1|Args1],
  Term2 =.. [Functor2|Args2],
  length(Args1, Arity),
  length(Args2, Arity),
  apply(Relation, [Functor1, Functor2, DegreeFunctor]),
  DegreeFunctor >= Lambda,
  compare_args(Args1, Args2, Relation, Lambda, DegreeArgs),
  Degree is min(DegreeFunctor, DegreeArgs).


% compare_args(?Args1, ?Args2, +Relation, +Lambda, ?Degree)

compare_args([], [], _Relation, _Lambda, 1).

compare_args([Arg1|MArgs1],[Arg2|MArgs2],Relation,Lambda,Degree) :-
  compare_terms(Relation, Arg1, Arg2, Lambda, DegreeArg),
  compare_args(MArgs1, MArgs2, Relation, Lambda, DegreeMArgs),
  Degree is min(DegreeArg, DegreeMArgs).
```

Code listing 4.7: Implementation of the term comparison algorithm.

only deserves to emphasize the use of higher-order predicates as `apply` in order to use the same algorithm with any of the fuzzy relations that the Bousi∼Prolog language has (excluding the fuzzy relation that participates in weak unification algorithm). The full code of the term comparison algorithm is found in the Code listing 4.7.

4.5.4. Closures computation

The computation of reflective, symmetric and/or transitive closure of fuzzy relations is a task performed by the predicate `ext_closure` of the BPL systemforeign library, written in C, since it requires the generation and management of tables that can get to be large and in that aspect C is much more efficient than Prolog.

From the proximity equations indicated by the programmer in the source code, the predicate `ext_closure` generates a bidimensional matrix with one row and one column for each symbol that intervenes in the relation. This matrix is called the *adjacency matrix*.

The adjacency matrix is initially filled with the approximation degrees of the proximity equations that partially specify the relation, after which the following algorithms are used to complete the matrix with the values that form the reflexive, symmetric and transitive closure of the relation (in this order) :

a) **Reflexive closure**. All the entries of the main diagonal of the adjacency matrix are set to 1 (the maximum possible approximation degree).

b) **Symmetric closure**. The entries of the adjacency matrix below to the main diagonal are copied to their respective symmetrical positions above to the diagonal and vice versa.

c) **Transitive closure**. A variant of the Warshall algorithm for binary adjacency matrices is applied, in which the Boolean operators *OR* and *AND* are respectively replaced by the maximum t-conorm and a t-norm chosen for the computation of transitivity.

The code of C method that is specifically responsible for calculating the closures of a fuzzy relation characterized by an adjacency matrix[6] can be found in the Code listing 4.8.

4.5.5. Syntactic analysis of terms

Based on what was decided during the design of the module *parser*, the parser of Bousi∼Prolog has been implemented through a defined clause grammar (DCG). Taking as a reference the formal Prolog specification collected in the ISO Prolog standard [14] and the Bousi∼Prolog syntax, in total about 60 rules have been nedeed to recognize all the constructions of both languages.

When compiling a DCG as the one developed in the BPL system, Prolog internally generates a series of predicates that build a parser for the language defined by the DCG rules. This parser, in accordance with the Prolog execution model, is a *top-down* parser with *backtracking* that reads the input strings from left to right [40].

Descending analyzers of this type, because of the need to use backtracking to fully rec-

[6]Since C does not have data structures for handling dynamic-sized arrays, for the development of the foreign library an auxiliary C module called *array* has been implemented that allows the creation of two-dimensional arrays of strings, numbers and pointers whose number of rows and columns may vary at runtime. The header file of this module defines a structure called `array` that contains all the necessary data about a dynamic matrix.

```
static void build_closure(array *pmAdjMatrix, int nSize,
                          int nClosureId, int nTNormId) {
  double dDegree, dTNormResult;
  bool bTransitive, bSymmetric, bReflexive;
  int i, j, k;

  /* extracts the closure properties from the closure id */
  /* [...] */

  /* reflexive closure */
  if(bReflexive) {
    for(i = 0; i < nSize; i++) {
      array_set_double(pmAdjMatrix, i, i, 1.0);
    }
  }
  /* symmetric closure */
  if(bSymmetric) {
    for(i = 0; i < nSize; i++) {
      for(j = 0; j < nSize; j++) {
        if(i != j && array_get_double(pmAdjMatrix, i, j) > 0.0) {
          array_set_double(pmAdjMatrix, j, i,
                           array_get_double(pmAdjMatrix, i, j));
        }
      }
    }
  }
  /* transitive closure (Warshall-like algorithm) */
  if(bTransitive) {
    for(k = 0; k < nSize; k++) {
      for(i = 0; i < nSize; i++) {
        for(j = 0; j < nSize; j++) {
          switch(nTNormId) {
          case TNM_MINIMUM:
            dTNormResult=MIN(array_get_double(pmAdjMatrix, i, k),
                             array_get_double(pmAdjMatrix, k, j));
            break;
          case TNM_PRODUCT:
            dTNormResult=PRODUCT(array_get_double(pmAdjMatrix,i,k),
                                 array_get_double(pmAdjMatrix,k,j));
            break;
          case TNM_LUKASIEWICZ:
            dTNormResult=LUKA(array_get_double(pmAdjMatrix, i, k),
                              array_get_double(pmAdjMatrix, k, j));
            break;
          }
          dDegree = MAX(array_get_double(pmAdjMatrix, i, j),
                        dTNormResult);
          array_set_double(pmAdjMatrix, i, j, dDegree);
        } /* for(j) */
      } /* for(i) */
    } /* for(k) */
  }
}
```

Code listing 4.8: Implementation of the algorithm to compute the reflexive, symmetric and/or transitive closure.

ognize input strings, are usually slightly slower than analyzers without backtracking. However, the biggest problem is that, if the original grammar is recursive to the left, the analyzer can fall into an infinite loop when recognizing certain chains [1].

Since the Prolog language has several left associative operators (`+`, `-`, `*` and, in general, all the arithmetic operators), its grammar collected in the standard ISO Prolog is recursive to the left. Just look at one of the productions of the non-terminal *left term* of the language syntax to check it:

***left term [n] = left term [n]** , operator [n / yfx] , term [n - 1]*

Therefore, in order to properly recognize programs written in Prolog (and Bousi∼Prolog) by means of a DCG, first, it is essential to eliminate the left recursion of the original grammar of the language, keeping its syntax intact. In particular, it is necessary to modify the way in which the Prolog terms are analyzed, which are those that may contain left associative operators.

To solve this problem, several options were posed, such as grouping the operators by priority and creating different rules for each group of them, or limiting the number of times that the same DCG rule could be applied recursively. However, while the first solution prevented the use of user-defined operators (a requirement of the system), the second solution arbitrarily restricted the length of the terms that could appear in the programs.

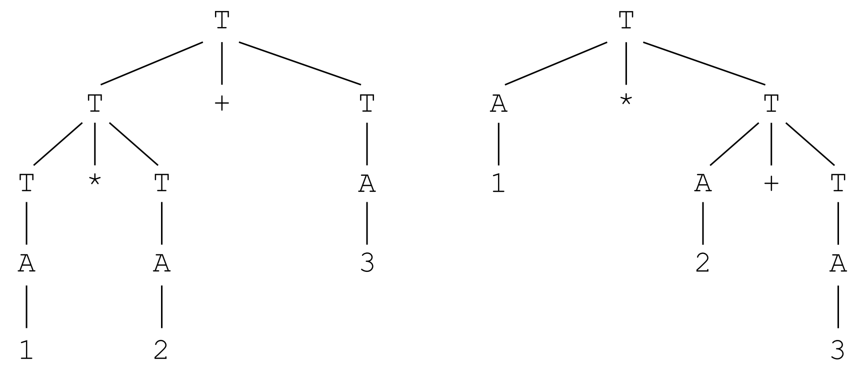
Finally, the solution that has been adopted is to recognize the Prolog terms by means of a right recursive grammar and generate flat lists with all the operands and operators (prefixes and infixes, not postfixes)[7] that appear in them. Then, these lists are passed to algorithms that generate the real syntactic tree of each term, based on the priority and the associativity of each of its operators.

The need to use additional algorithms to obtain the real syntactic tree of the terms from the result of the right recursive analyzer is reflected in the example of Figure 4.15. In this figure, it is analyzed the expression `1 * 2 + 3` using two analyzers: one left-right recursive and another recursive only to the right. While the first one is able of generating the correct syntactic tree of the expression, using the precedence of its operators, with the second one we obtain a tree that does not represent the expected expression.

To see how the syntactic analysis has finally been implemented, an example will be

[7]Not allowing the use of postfix operators does not represent a major limitation because the standard ISO Prolog does not define any default operator of that type. However, it prevents users from declaring their own postfix operators.

Left-right recursive analyzer	right recursive analyzer
`T → T + T`	`T → A + T`
`T → T * T`	`T → A * T`
`T → A`	`T → A`
`A → 1 | 2 | 3`	`A → 1 | 2 | 3`
Precedence: +, *	
Analyzed expression:	Analyzed expression:
`1 * 2 + 3`	`1 * 2 + 3`



`(1 * 2) + 3`	`1 * (2 + 3)`
Correct	*Incorrect*

Figure 4.15: Limitations of right recursive analyzers.

shown starting from the following Prolog term:

```
X is 5 + -2, true, fail
```

The DCG rules of the module *parser* convert this term into a list that contains all its operands and operators in the order in which they appear in the source code. During this stage it is checked that the term is correct syntactically (for example, it is verified that there are not two consecutive infix operators). For the proposed term the list that would be obtained would be:

```
[X, operator(is, infix), 5, operator(+, infix),
 operator(-, prefix), 2, operator(',', infix),
 true, operator(',', infix), fail]
```

This list is passed to algorithms developed expressly for the BPL system that apply the well-known “divide and conquer” technique to obtain the syntactic tree associated with the initial term. Since all operators have a well defined priority and associativity, it will only be possible to generate a term from each list.

If there were ambiguities in the construction of the syntactic tree (for example, because several non-associative operators or incompatible associativities would have been the same level), an error at compile time would occur.

Next is presented the pseudocode of the algorithms that have been developed to create the syntactic tree of a term from a list generated by the rules of the DCG. The two algorithms are implemented within the module *parser*.

ALGORITHM 1. generateSyntacticTree

INPUT: *list*, a flat list with the operands and operators of a term.

OUTPUT: a syntactic tree representing the term associated with *list*.

```
Begin
If list only has one element then
    tree ← only element of list
else
    operator ← higherPriorityOperator(list)
    LeftList ← sublist of list with the elements prior to operator
    RigthList ← sublist of list with the elements after operator
    If operator is an infix operator then
        LeftExpression ← generateSyntacticTree(LeftList)
        RightExpression ← generateSyntacticTree(RigthList)
        tree ← $\boxed{\begin{array}{c} operator \\ / \quad \backslash \\ LeftExpression \quad RightExpression \end{array}}$
    Else
        If LeftList is not empty then Throw error
        Expression ← generateSyntacticTree(RigthList)
        tree ← $\boxed{\begin{array}{c} operator \\ | \\ RightExpression \end{array}}$
    End If
End If
Return tree
```

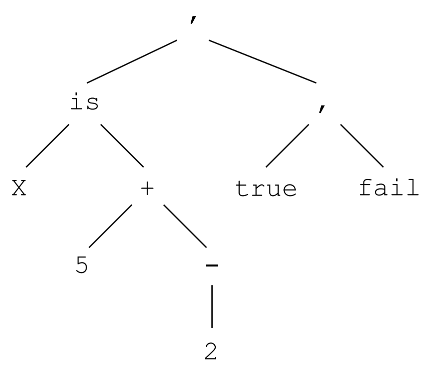
ALGORITHM 2. higherPriorityOperator

INPUT: *list*, a flat list with the operands and operators of a term.

OUTPUT: main operator of the term passed as a parameter. The main operator is the highest priority operator of a term or, if there are several with the same priority, the one with the highest level according to the rules of associativity.

```
Begin
mainOperator ← null
higherPriority ← –1
For each element operator of list do
    If operator is an operator then
        priority ← priority of operator
        If priority > higherPriority then
            // New operator of higher priority
            higherPriority ← priority
            mainOperator ← operator
        Else If priority = higherPriority then
            If operator is left associative (yfx class) then
                // Left associativity, change the current operator
                higherPriority ← priority
                mainOperator ← operator
            Else If operator is not left associative (xfy/xfx/fx/fy class)
            and mainOperator is right associative (xfy/fy class) then
                // Right associativity, maintain the current operator
            Else
                Throw error Collision of operators
            End If
        End If
    End If
End For
Return mainOperator
```

Applying the algorithms explained to the list generated from the example term, it would result the following syntactic tree, which reflects the real structure of the initial term.



```
X is 5 + -2, true, fail
```

4.5.6. Execution of higher-order predicates

Although the translation of BPL code to TPL code does not affect the behavior of the vast majority of the predefined predicates of the underlying Prolog system (in our case SWI-Prolog), there is a series of predefined predicates that require special treatment so that they can be executed correctly in the BPL system.

These predicates that need to be treated differently from the others are within the so-called *higher-order predicates*, those that can receive as parameters one or several objectives composed of calls to other predicates.

To see the problems that arise when executing this class of predicates, consider the following Prolog program that makes use of the higher-order predicate `findall`, which finds all the solutions to the goal indicated in the second argument:

```
search(L) :- findall(X, p(X), L).
p(1).  p(2).
```

In this example the predicate `search` would unify the variable `L` with the list `[1, 2]`, which are the only two values for which `p(X)` holds.

Now, assume that the previous code is entered into a BPL program and the compiler translates it into TPL code exactly as explained so far. The generated TPL code would be (after some simplifications) as follows:

```
prog_search(A1, D) :- unify_arguments([[A1, L, D1]]),
                      findall(X, p(X), L), min_degree([D1], D).
prog_p(A1, D) :- unify_arguments([[A1, 1, D1]]),
                 min_degree([D1], D).
prog_p(A1, D) :- unify_arguments([[A1, 2, D1]]),
                 min_degree([D1], D).
```

If this program were loaded with the command processor of the BPL system and an attempt to launch the query `search(L)`, no result would be obtained, since there is no predicate `p` of arity 1 in the TPL code. Instead, during the compilation process the original predicate `p` has been converted into a predicate called `prog_p` that includes an additional output parameter (the approximation degree).

Therefore, the correct translation to TPL code of the predicate `search` of the example would be the one shown below:

```
prog_search(A1, D) :- unify_arguments([[A1, L, D1]]),
                      findall(X, prog_p(X, _), L),
                      min_degree([D1], D).
```

With respect to this translation it is convenient to make two clarifications. First, the approximation degree of `prog_p` is ignored, since only the approximation degrees of the predicates that appear in the body of the clauses are considered relevant. Secondly, the translation has been done at compile time and not at run time, in order to transfer as much "logic" as possible to the compiler and thus accelerate the execution process.

To carry out this special translation the compiler of the BPL system maintains a list of higher-order predicates with the type of each of its arguments. When a predicate that appears in that list is recognized, the parameters that contain goals are translated as if they were a clause of the program, which entails adding the degree variables and the program prefix to the user-defined predicates.

However, in addition to the higher order predicates, it is also possible to use the term construction operators, such as `=..` to compose goals at run time. See for example the following Prolog code fragment, in which the variable `Pred` is unified with a certain term and then launched as a goal:

```
search(Name, Value) :- Pred =..  [Name, Value], Pred.
```

If this code were translated into TPL, the same problem would occur as in the previous cases, since the degree variable is not added to the goal stored in `Pred`. For this reason, in the body of the clauses of the BPL programs, isolated variables are not allowed to appear. These should always appear within a higher-order predicate as `call`[8], which internally is translated to a call to an auxiliary predicate of the module *evaluator* called `bpl_call`.

Finally, we must also mention the special treatment of the higher-order predicates

[8]Given any variable `X`, the standard ISO Prolog states that executing `X` and `call(X)` must produce exactly the same result.

`assert` and `retract`, which allow respectively adding and removing clauses (in this case only facts)[9] of the programs. In the same way that to compile a BPL file it is necessary to perform an expansion of the rules, when adding and eliminating clauses dynamically the same criteria must be followed.

So, given the following BPL code:

```
young ∼ old = 0.2.
new :- assert(young(carlos)).
```

The translation that would be made of the predicate `new` would be:

```
prog_new(D)  :-
     assert( (prog_young(A1,D1):-unify_arguments([[A1,carlos,D11]]),
                                  min_degree([D11],D1)) ),
     assert( (prog_old(A1,D1):-over_lambdacut(0.2),
                                unify_arguments([[A1,carlos,D11]]),
                                min_degree([D11,0.2],D1)) ),
     min_degree([], D).
```

In summary, the BPL system is compatible with all the higher-order predicates that were collected in the requirements specification (see Page 48), also including the two types of negations available in the Bousi∼Prolog language (`not` and `\+`).

4.5.7. Library of foreign predicates

The dynamic library of foreign predicates that uses the BPL system consists of a total of eight modules implemented in C, which are briefly described below.

- **closure.c**. This module is responsible for computing the closure of a fuzzy relation defined by means of a set of proximity equations.
- **fuzzysets.c**. The module *fuzzysets* implements the generation of a reflexive fuzzy relation from a list of fuzzy subsets.
- **lexical.c**. This file is automatically generated by the Flex tool from the file *scanner.lex*, which contains the definition of the lexical analyzer of the Bousi∼Prologlanguage.
- **tokenize.c**. The module *tokenize* has a foreign predicate that serves as an interface between Prolog and the lexical analyzer generated by Flex.

[9]According to the standard ISO Prolog, the predicates `assert` and `retract` must be able to be used to add and remove both facts and rules of the programs. However, adding new rules to a program at runtime results in a self-modifying code, a technique not recommended and banished in modern programming languages [40, 15]. On the other hand, the controlled use of `assert` and `retract` with facts can be useful for maintaining a dynamic database or declaring global variables.

- **shell.c**. In this source code file all external predicates related to command line manipulation are placed. The predicates of this module make use of the functions offered by the *Editline* library.
- **leditwin.c**. This module contains alternative implementations for Windows of some methods of the *Editline* library. Most of the functions of this module have been extracted from the *Editline* and *Linenoise* libraries.
- **array.c**. The module *array* is an auxiliary module that allows to create vectors and matrices of variable size in C, and has been created with the purpose of abstracting to the rest of the modules from the memory management tasks for this type of structures.
- **install.c**. In this source file is found the installation function of the foreign predicates that SWI-Prolog needs to load a foreign library.

4.5.8. Example sessions

In the Figures 4.16 to 4.24 of the following pages several example sessions of the BPL system are shown. They show the developed system in execution, carrying out some of the most relevant use cases that were collected in the analysis phase.

In the execution of the chosen cases of use, the compiler, the loader/interpreter and, of course, the command processor of the BPL systemintervene, so everything that was explained in this and the previous sections is illustrated.

4.6. TESTING

To ensure the quality of the developed code, the implementation phase must be accompanied by a testing phase in which the correct operation of the built modules is verified.

This section aims to explain the approach followed to elaborate the automated functional tests of the BPL system, and present a report with the code coverage reached by these tests.

4.6.1. Test plan

Being an interpreter of a programming language, the interaction between the users and the BPL system will consist mainly of loading programs in memory and then executing a series of queries and seeing its results. The test plan of BPL system has been designed keeping in mind that this will be the main scenario of use of the system.

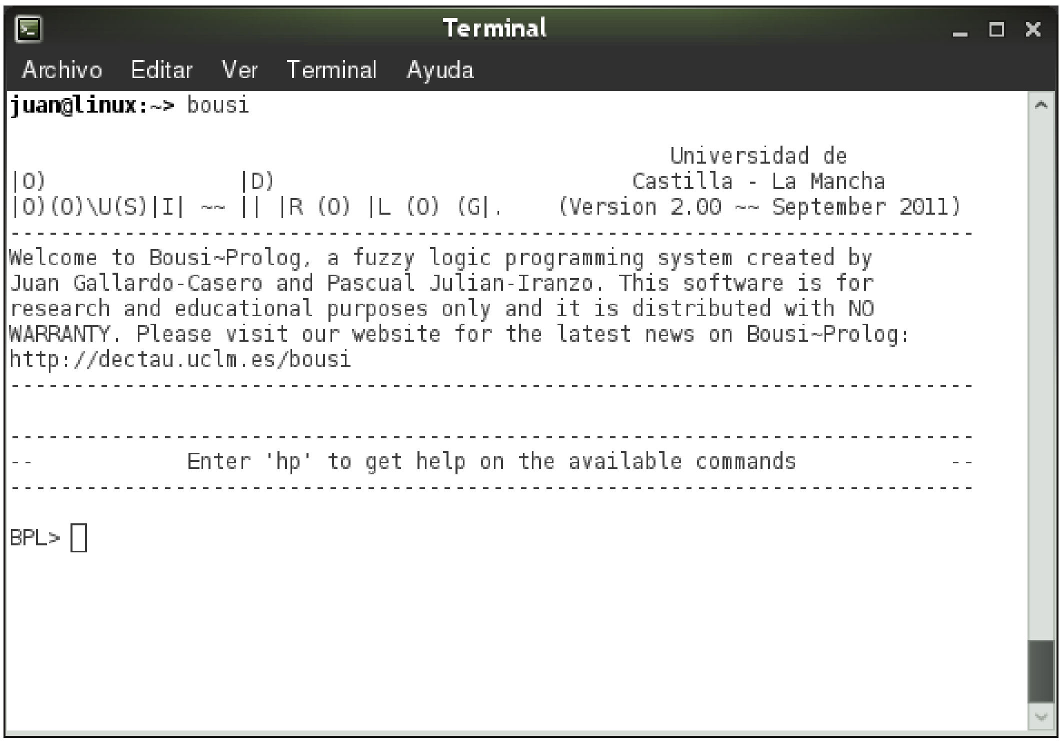
In this way, most of the BPL system tests are small Prolog or Bousi∼Prolog code fragments that are loaded into the BPL system and then run to see if the results offered by the system are the expected. By choosing an appropriate set of code fragments, with this approach it is possible to cover almost the entire system code (except for the subsystem of the command processor).

In more detail way, the BPL system tests can be divided into four large blocks:

- **Testing of Prolog predicates**. They intend to ensure that all predefined Prolog predicates that were mentioned in the requirements specification (see Page 48) work correctly under the BPL system.

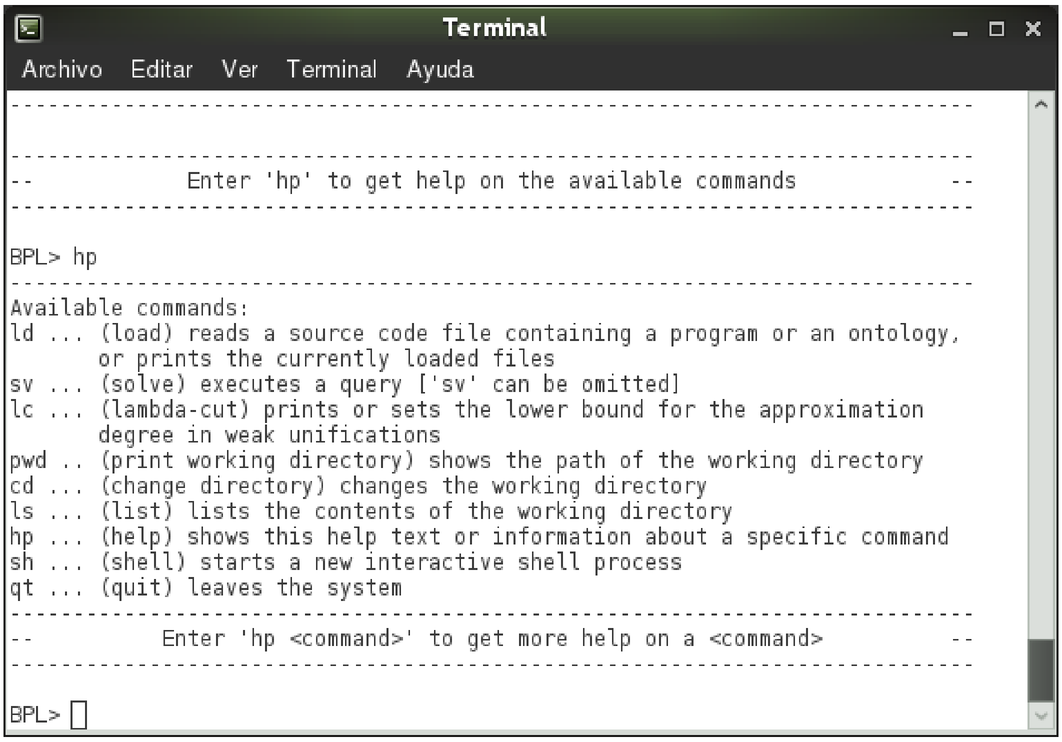
  Based on the fact that most of the Prolog predicates indicated in the requirements belong to the standard ISO Prolog, the examples of each predicate that appear in the standard itself [14] have been used to test its behavior.

  Therefore, the tests of each Prolog predicate consist in executing their corresponding examples on SWI-Prolog and the BPL system, and then comparing the solutions and the output obtained in both executions. In total, more than 350 examples extracted from the documentation of the standard ISO Prologhave been added to the tests.



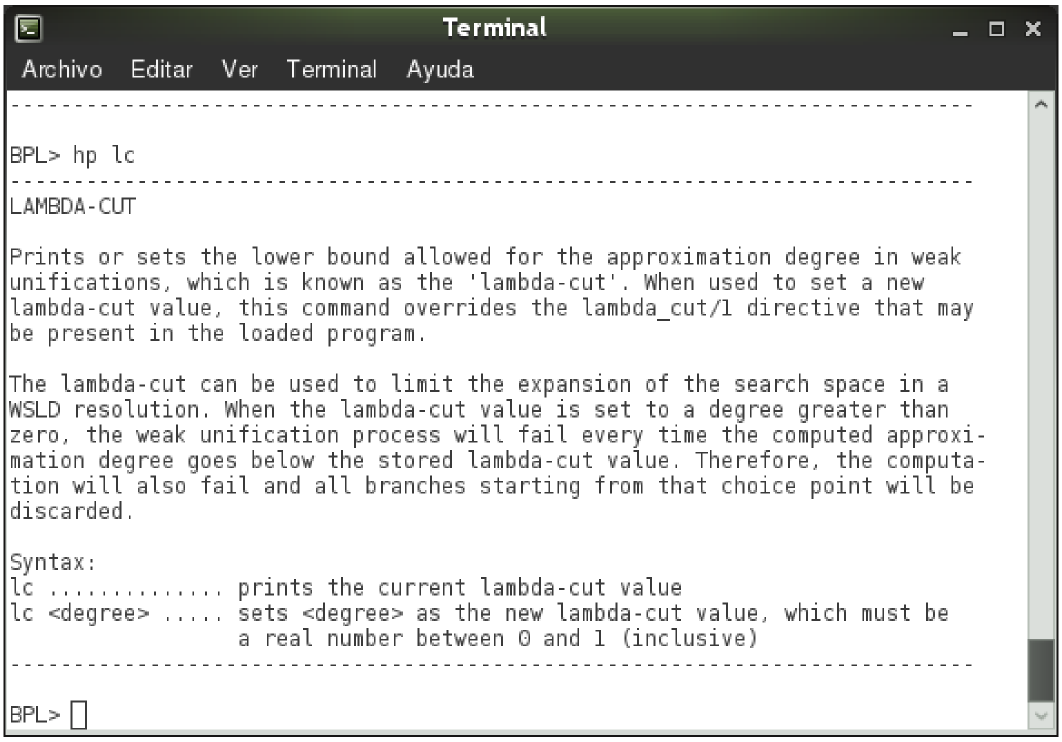
```
juan@linux:~> bousi

                                                    Universidad de
|O)               |D)                            Castilla - La Mancha
|O)(O)\U(S)|I| ~~ || |R (O) |L (O) (G|.    (Version 2.00 ~~ September 2011)
----------------------------------------------------------------------------
Welcome to Bousi~Prolog, a fuzzy logic programming system created by
Juan Gallardo-Casero and Pascual Julian-Iranzo. This software is for
research and educational purposes only and it is distributed with NO
WARRANTY. Please visit our website for the latest news on Bousi~Prolog:
http://dectau.uclm.es/bousi
----------------------------------------------------------------------------

----------------------------------------------------------------------------
--            Enter 'hp' to get help on the available commands            --
----------------------------------------------------------------------------

BPL> 
```

Figure 4.16: Welcome to the BPL system.



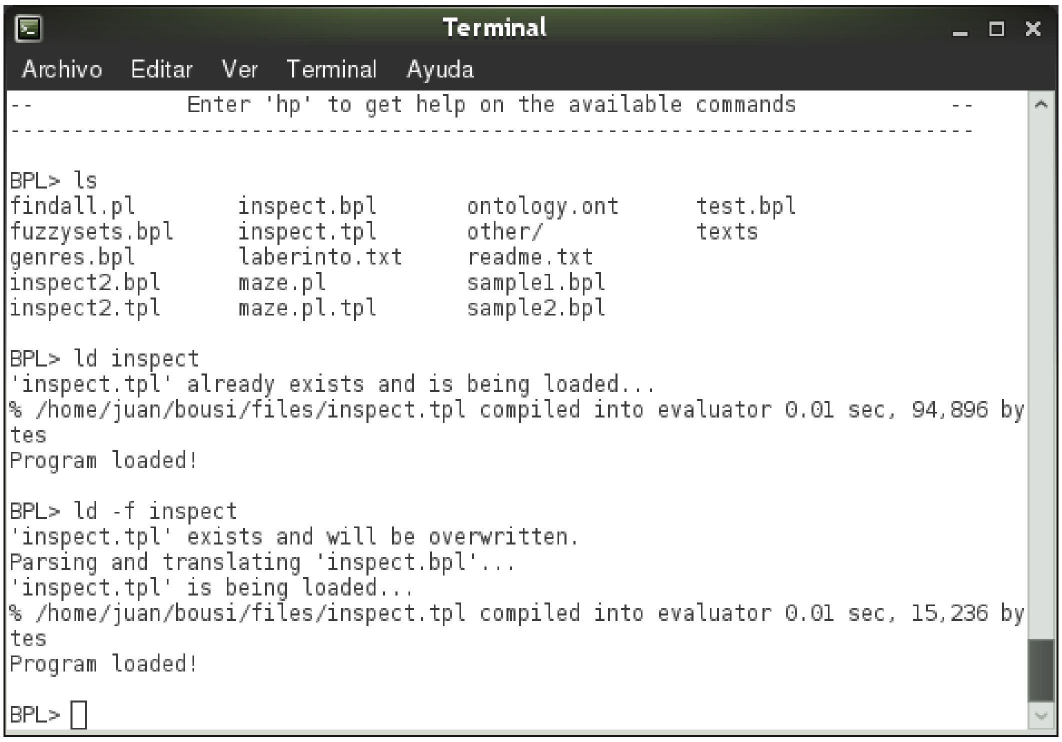
```
---------------------------------------------------------------------------


---------------------------------------------------------------------------
--             Enter 'hp' to get help on the available commands          --
---------------------------------------------------------------------------

BPL> hp
---------------------------------------------------------------------------
Available commands:
ld ... (load) reads a source code file containing a program or an ontology,
       or prints the currently loaded files
sv ... (solve) executes a query ['sv' can be omitted]
lc ... (lambda-cut) prints or sets the lower bound for the approximation
       degree in weak unifications
pwd .. (print working directory) shows the path of the working directory
cd ... (change directory) changes the working directory
ls ... (list) lists the contents of the working directory
hp ... (help) shows this help text or information about a specific command
sh ... (shell) starts a new interactive shell process
qt ... (quit) leaves the system
---------------------------------------------------------------------------
--           Enter 'hp <command>' to get more help on a <command>        --
---------------------------------------------------------------------------

BPL>
```

Figure 4.17: Getting help on the BPL system.



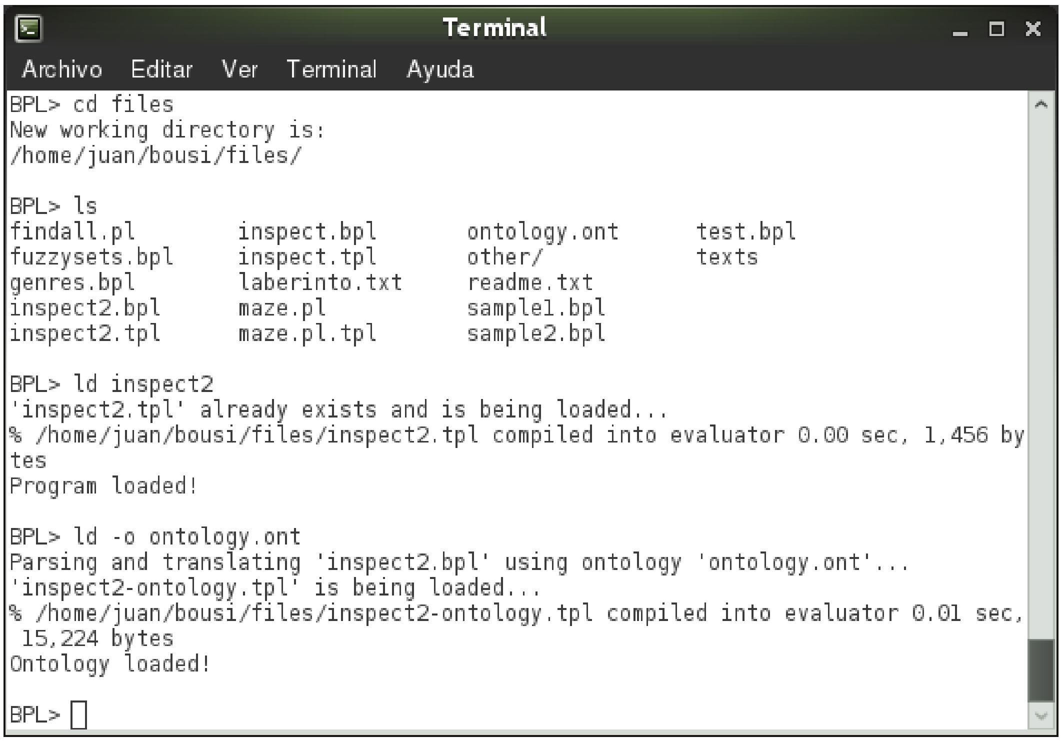
```
---------------------------------------------------------------------------

BPL> hp lc
---------------------------------------------------------------------------
LAMBDA-CUT

Prints or sets the lower bound allowed for the approximation degree in weak
unifications, which is known as the 'lambda-cut'. When used to set a new
lambda-cut value, this command overrides the lambda_cut/1 directive that may
be present in the loaded program.

The lambda-cut can be used to limit the expansion of the search space in a
WSLD resolution. When the lambda-cut value is set to a degree greater than
zero, the weak unification process will fail every time the computed approxi-
mation degree goes below the stored lambda-cut value. Therefore, the computa-
tion will also fail and all branches starting from that choice point will be
discarded.

Syntax:
lc .............. prints the current lambda-cut value
lc <degree> ..... sets <degree> as the new lambda-cut value, which must be
                  a real number between 0 and 1 (inclusive)
---------------------------------------------------------------------------

BPL>
```

Figure 4.18: Getting help on the *lc* command in the BPL system.



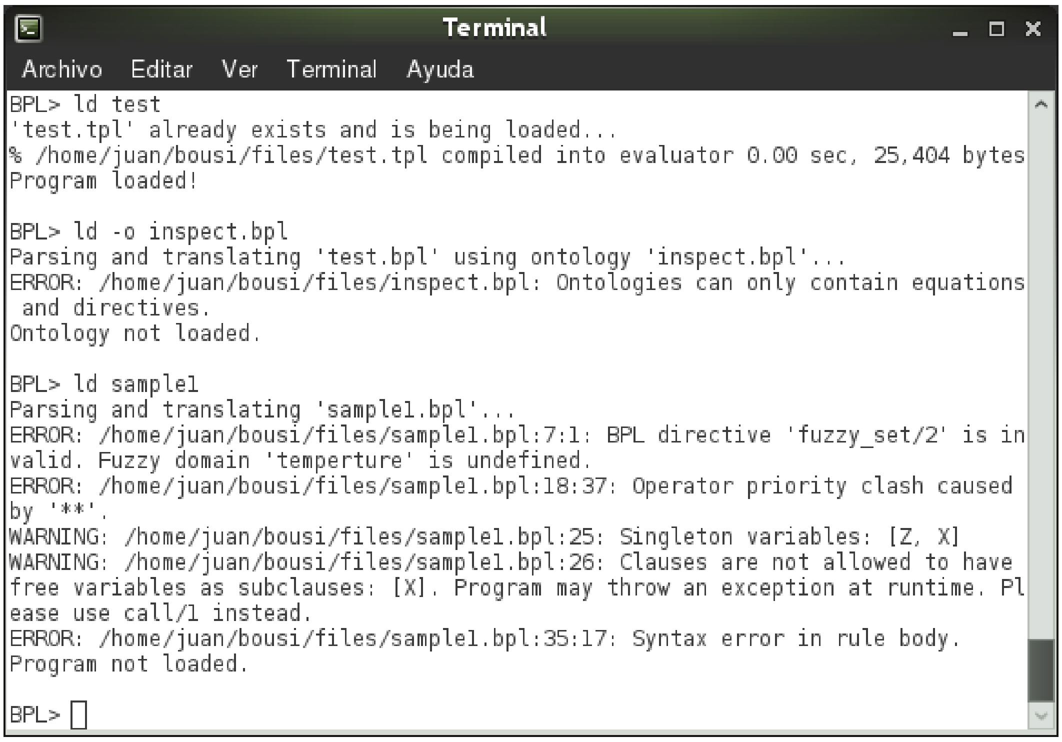
```
--           Enter 'hp' to get help on the available commands           --
---------------------------------------------------------------------------

BPL> ls
findall.pl        inspect.bpl       ontology.ont      test.bpl
fuzzysets.bpl     inspect.tpl       other/            texts
genres.bpl        laberinto.txt     readme.txt
inspect2.bpl      maze.pl           sample1.bpl
inspect2.tpl      maze.pl.tpl       sample2.bpl

BPL> ld inspect
'inspect.tpl' already exists and is being loaded...
% /home/juan/bousi/files/inspect.tpl compiled into evaluator 0.01 sec, 94,896 by
tes
Program loaded!

BPL> ld -f inspect
'inspect.tpl' exists and will be overwritten.
Parsing and translating 'inspect.bpl'...
'inspect.tpl' is being loaded...
% /home/juan/bousi/files/inspect.tpl compiled into evaluator 0.01 sec, 15,236 by
tes
Program loaded!

BPL> 
```

Figure 4.19: Loading a program in the BPL system.



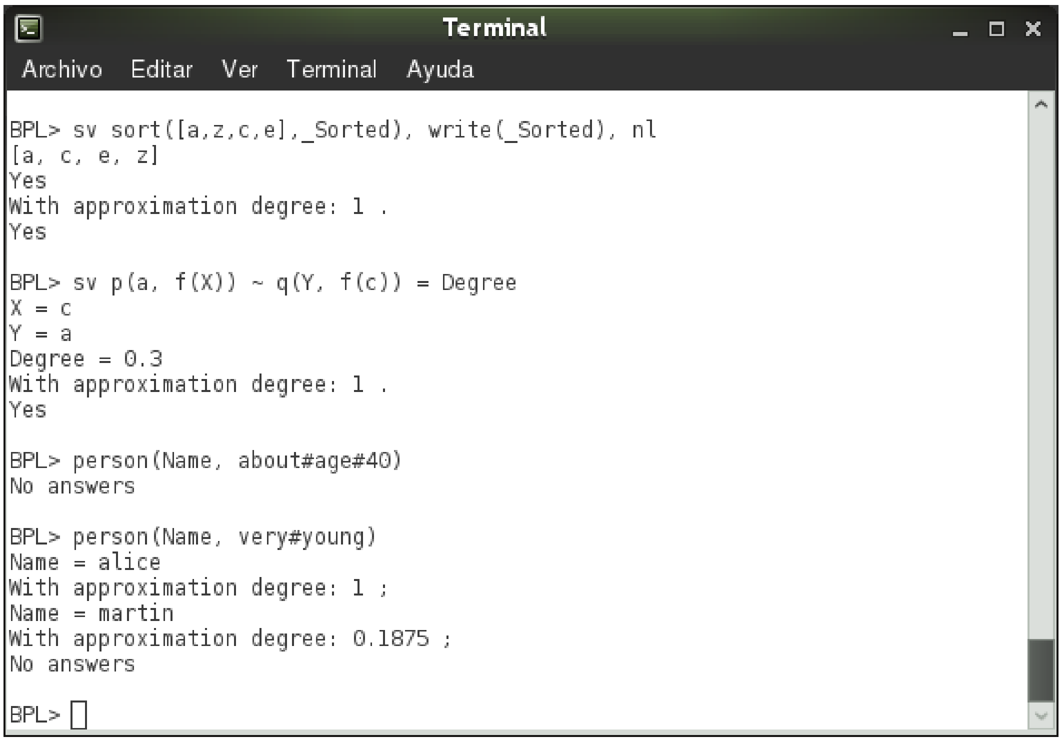
```
BPL> cd files
New working directory is:
/home/juan/bousi/files/

BPL> ls
findall.pl        inspect.bpl       ontology.ont      test.bpl
fuzzysets.bpl     inspect.tpl       other/            texts
genres.bpl        laberinto.txt     readme.txt
inspect2.bpl      maze.pl           sample1.bpl
inspect2.tpl      maze.pl.tpl       sample2.bpl

BPL> ld inspect2
'inspect2.tpl' already exists and is being loaded...
% /home/juan/bousi/files/inspect2.tpl compiled into evaluator 0.00 sec, 1,456 by
tes
Program loaded!

BPL> ld -o ontology.ont
Parsing and translating 'inspect2.bpl' using ontology 'ontology.ont'...
'inspect2-ontology.tpl' is being loaded...
% /home/juan/bousi/files/inspect2-ontology.tpl compiled into evaluator 0.01 sec,
 15,224 bytes
Ontology loaded!

BPL> 
```

Figure 4.20: Loading an ontology in the BPL system.



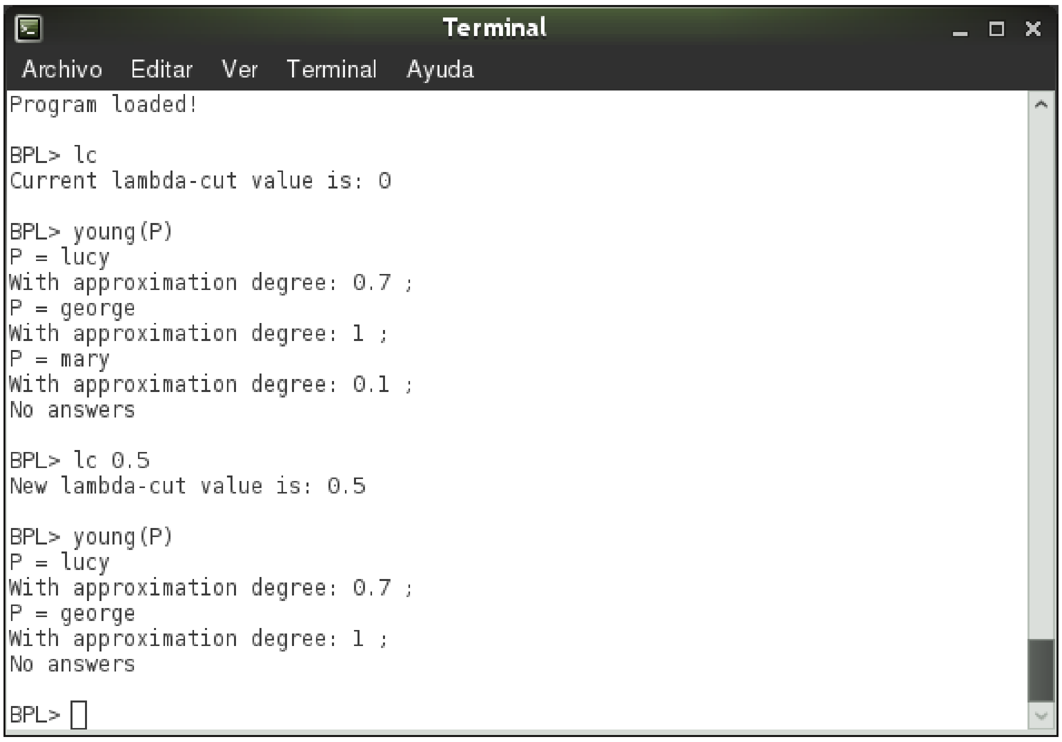
```
BPL> ld test
'test.tpl' already exists and is being loaded...
% /home/juan/bousi/files/test.tpl compiled into evaluator 0.00 sec, 25,404 bytes
Program loaded!

BPL> ld -o inspect.bpl
Parsing and translating 'test.bpl' using ontology 'inspect.bpl'...
ERROR: /home/juan/bousi/files/inspect.bpl: Ontologies can only contain equations
 and directives.
Ontology not loaded.

BPL> ld sample1
Parsing and translating 'sample1.bpl'...
ERROR: /home/juan/bousi/files/sample1.bpl:7:1: BPL directive 'fuzzy_set/2' is in
valid. Fuzzy domain 'temperture' is undefined.
ERROR: /home/juan/bousi/files/sample1.bpl:18:37: Operator priority clash caused
by '**'.
WARNING: /home/juan/bousi/files/sample1.bpl:25: Singleton variables: [Z, X]
WARNING: /home/juan/bousi/files/sample1.bpl:26: Clauses are not allowed to have
free variables as subclauses: [X]. Program may throw an exception at runtime. Pl
ease use call/1 instead.
ERROR: /home/juan/bousi/files/sample1.bpl:35:17: Syntax error in rule body.
Program not loaded.

BPL> 
```

Figure 4.21: Loading a code file with errors in the BPL system.



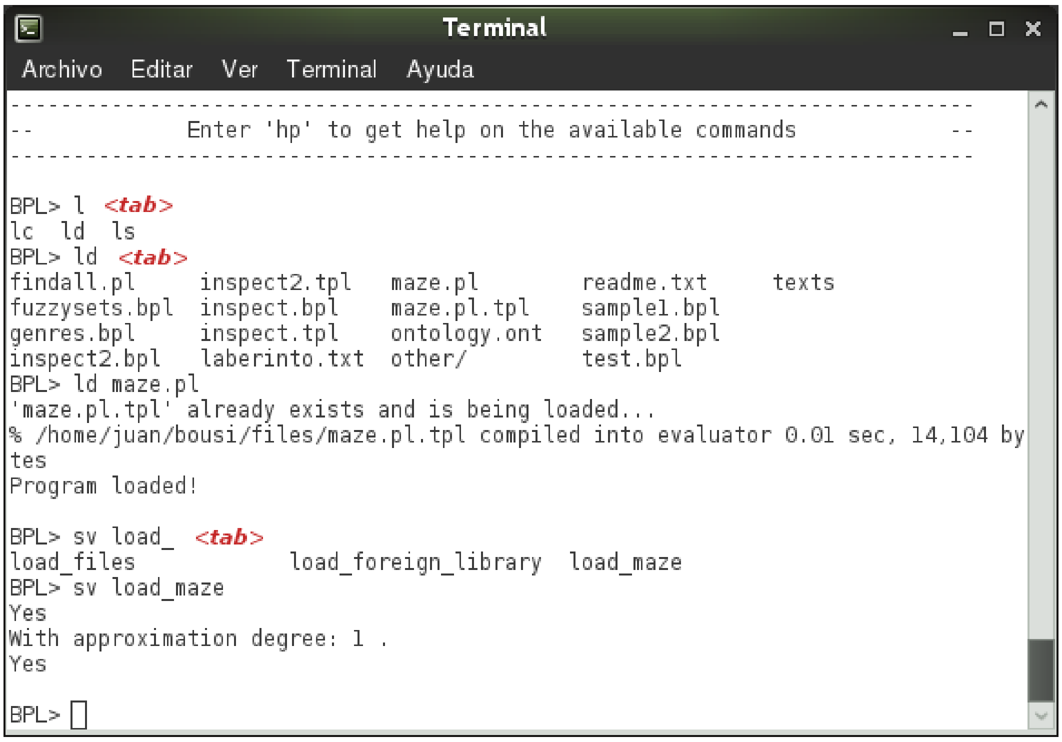
```
BPL> sv sort([a,z,c,e],_Sorted), write(_Sorted), nl
[a, c, e, z]
Yes
With approximation degree: 1 .
Yes

BPL> sv p(a, f(X)) ~ q(Y, f(c)) = Degree
X = c
Y = a
Degree = 0.3
With approximation degree: 1 .
Yes

BPL> person(Name, about#age#40)
No answers

BPL> person(Name, very#young)
Name = alice
With approximation degree: 1 ;
Name = martin
With approximation degree: 0.1875 ;
No answers

BPL> 
```

Figure 4.22: Execution of queries in the BPL system.



```
Program loaded!

BPL> lc
Current lambda-cut value is: 0

BPL> young(P)
P = lucy
With approximation degree: 0.7 ;
P = george
With approximation degree: 1 ;
P = mary
With approximation degree: 0.1 ;
No answers

BPL> lc 0.5
New lambda-cut value is: 0.5

BPL> young(P)
P = lucy
With approximation degree: 0.7 ;
P = george
With approximation degree: 1 ;
No answers

BPL> 
```

Figure 4.23: Querying and modifying the *lambda cut* in the BPL system.



```
---------------------------------------------------------------------------
--            Enter 'hp' to get help on the available commands           --
---------------------------------------------------------------------------

BPL> l  <tab>
lc  ld  ls
BPL> ld  <tab>
findall.pl     inspect2.tpl   maze.pl        readme.txt     texts
fuzzysets.bpl  inspect.bpl    maze.pl.tpl    sample1.bpl
genres.bpl     inspect.tpl    ontology.ont   sample2.bpl
inspect2.bpl   laberinto.txt  other/         test.bpl
BPL> ld maze.pl
'maze.pl.tpl' already exists and is being loaded...
% /home/juan/bousi/files/maze.pl.tpl compiled into evaluator 0.01 sec, 14,104 by
tes
Program loaded!

BPL> sv load_  <tab>
load_files          load_foreign_library  load_maze
BPL> sv load_maze
Yes
With approximation degree: 1 .
Yes

BPL> 
```

Figure 4.24: BPL system command processor autocomplete.

- **Testing of Bousi∼Prolog predicates**. They serve to check the operation of all directives and specific sentences of the Bousi∼Prolog language.

  In this case, the approach followed to verify the predicates of the Prolog language is unfeasible, so the tests consist of a set of Bousi∼Prolog code files in which several test predicates are defined in order to cover the largest number of possible situations.
- **Testing of wrong programs**. They allow to ensure the correct behavior of the BPL system when loading a program that has errors or warnings.

  To perform these tests, several Bousi∼Prolog code files have been created with syntax errors, invalid directives, unused variables (*singleton* variables), etc. During the execution of the tests, attempts are made to load these files into memory and it is checked if the processor shows error or warning messages only on the lines where there is really a problem.
- **Testing the *shell***. They are used to confirm that all command processor commands work properly and that invalid commands are detected.

  In this block of tests, a series of predetermined commands are executed following the same line that would be followed if the user had entered those commands in the terminal. After executing each command, the output generated by the processor is analyzed to see if it contains the expected messages[10].

  Given its dependency on the operating system, certain commands from the command processor, such as `ls` or `qt`, have been excluded from the tests.

4.6.2. Coverage report

The most recent versions of SWI-Prolog include a library called *test_cover*, through which it is possible to analyze which clauses of an application are used (they are «covered») by a set of tests.

The report that *test_cover* generates only measures the coverage of clauses, that is, the percentage of clauses in the application that have been executed and successfully completed at least once during execution of all the tests.

Table 4.2 shows the coverage achieved by the tests in each module of the BPL system, as well as the global coverage of the entire system. As it can be seen, in all the modules a

[10]For example, for the command `lc 1.5` an error message is expected, while the `ld` command should show the full path of the currently loaded program.

	Coverage reached
Module *bplHelp*	100%
Module *bplShell*	85%
Module *directives*	100%
Module *evaluator*	95%
Module *flags*	96%
Module *parser*	98%
Module *translator*	100%
Module *utilities*	96%
TOTAL	**95%**

Table 4.2: Report of coverage of the BPL systemtests.

coverage exceeding 95% has been achieved except in *bplShell*, where the coverage is 85% due to the fact that several commands have had to be excluded from the tests.

On the other hand, the module *foreign* is not included in the previous table, because the library *test_cover* does not measure the coverage of foreign predicates (which does not mean that they are not executed by the tests), nor the module *bousi*, because this module only has the predicate that acts as the entry point of the application implementing the BPL system.

CHAPTER 5. BOUSI∼PROLOG APPLICATIONS

The goal of this chapter is to present a series of practical examples in which the Bousi∼Prolog programming language and, in particular, the implementation that has been done in this work, can be applied successfully.

In [22, 20, 38] you can find more examples of the Bousi∼Prolog language use, as well as a more detailed explanation of several of the examples presented in this chapter[1].

5.1. TEXT CATALOGING

The BPL system that has been developed throughout this project has been used in [34] to implement a declarative approach to text cataloging, making use of the flexible matching of terms and the similarity relations that offers Bousi∼Prolog.

In the proposal of the aforementioned article, a BPL program is used to perform a flexible search of the terms of a document that are similar to the categories by which it is to be classified. Furthermore, the post-processing of the results obtained in the search is also carried out with the same program.

The Code listing 5.1 contains a code fragment analogous to the one used by the text classifier to search the words of a document that are close to a certain concept. In the real application the terms to be analyzed are read from some text files, but for simplicity in the presented example they are passed in a list within the program itself.

The predicate in charge of looking for the similar terms is `search_term`, which receives as a parameter the concept to be searched and the list of words in the input document. This predicate returns a new list with elements of the type `t(Word, Degree)`, where `Word` is a word similar to the searched concept and `Degree` indicates the degree of approximation between both terms.

Once the search is completed, the list returned by `search_term` is passed to the `group_results` predicate to count and save the number of times each found term is repeated, thus obtaining the results that are passed to the next phase of the classifier.

To determine if two terms are close to each other, in [34] is proposed to use an ontology

[1]Those examples were written for the low-level implementation of Bousi∼Prolog and have been rewritten to conform to certain syntax changes and take advantage of features offered by the high-level implementation, such as customized fuzzy relations or compatibility with higher-order predicates.

```
% search_term(+Term, +WordList, -Results)
%   Looks for words similar to Term in WordList and returns a list
%   with a t(SimilarTerm, Degree) item for each word found.

search_term(_Term, [], []).

search_term(Term, [Word|MoreWords], [t(Word, Degree)|Results]) :-
  Term ~1~ Word = Degree, !,
  search_term(Term, MoreWords, Results).

search_term(Term, [_Word|MoreWords], Results) :-
  search_term(Term, MoreWords, Results).


% group_results(+Results, -GroupedResults)
%   Removes the duplicate items from Results and adds the number of
%   occurrences to each item. Given [t(X,0.5), t(Y,0.8), t(X,0.5)],
%   this predicate will return [t(X,2,0.5), t(Y,1,0.8)].

group_results(Results, GroupedResults) :-
  setof(t(Term, Occurrences, Degree),
        (member(t(Term, Degree), Results),
         count_results(Results, Term, Degree, Occurrences)),
        GroupedResults).


% count_results(+Results, +Term, +Degree, -Occurrences)
%   Counts how many t(Term, Degree) items are in the Results list.

count_results(Results, Term, Degree, Occurrences) :-
  bagof(t(Term, Degree),
        member(t(Term, Degree), Results),
        FilteredList),
  length(FilteredList, Occurrences).


% sample_search(+Term, -Results)
%   Runs a sample search for the given Term.

sample_search(Term, Results) :-
  search_term(Term, [agriculture, department, report, farm,
                     own, reserve, national, average, price,
                     loan, release, price, reserves, matured,
                     bean, grain, enter, corn, sorghum, rates,
                     bean, potato],
              UngroupedResults),
  group_results(UngroupedResults, Results).
```

Code listing 5.1: Bousi∼Prolog Program for a flexible search of terms.

```
% Ontology definition

:- fuzzy_rel(~1~, [reflexive, symmetric, transitive(min)]).

wheat ~1~ bean = 0.315.
wheat ~1~ corn = 0.315.
wheat ~1~ grass = 0.315.
wheat ~1~ horse = 0.315.
wheat ~1~ human = 0.205.
bean ~1~ crop = 0.315.
bean ~1~ corn = 0.48.
bean ~1~ child = 0.33.
bean ~1~ grass = 0.315.
bean ~1~ horse = 0.335.
bean ~1~ flower = 0.315.
bean ~1~ animal = 0.35.
bean ~1~ potato = 0.5.
bean ~1~ table = 0.35.
```

Code listing 5.2: Ontology used to test the text classifier [34].

modeled through a proximity or similarity relation. However, this ontology has not been deliberately included in the Code listing 5.1, since one of the advantages of the BPL system is that it allows the separate loading of a program and an ontology, in order to use the same program with different ontologies and compare their results.

The Code listing 5.2 shows one of the ontologies used in [34] to test the text classifier, which in this case represents the «structurally analogous» to `wheat`. The proximity equations of this ontology have been obtained through ConceptNet note, a commonsense knowledge base freely available, and then manually translated into Bousi∼Prolog syntax.

Once the Code listing 5.1 program and the Code listing 5.2 ontology are loaded in the BPL system, queries can be launched to see how many terms, close to one given, are in the sample document contained in the `sample_search` predicate, which has been extracted from the Reuters document collection[2]. For example, for the concept `corn`, the result obtained is as follows:

```
BPL> sample_search(corn, Results)
Results = [t(bean,2,0.48),t(corn,1,1.0),t(potato,1,0.48)]
With approximation degree:  1
```

As you can see, the system returns that the term `potato` is similar to `corn` although there is no equation that indicates this in the ontology because the BPL system automatically generates the transitive closure of the relation specified.

[2]http://www.daviddlewis.com/resources/testcollections/reuters21578/

5.2. FLEXIBLE DEDUCTIVE DATABASES

Databases are systems that are used to maintain and manage large amounts of information. When this information comes from the real world, database systems should offer mechanisms to deal with the vagueness and imprecision that often appear when modeling reality. The incorporation of this type of techniques to the database managers has given rise to what is known as *flexible databases*.

Within the field of flexible databases, one of the approaches studied is to include concepts from fuzzy logic and fuzzy set theory as a way to handle uncertainty and vagueness. The Buckles-Petry and Shenoi-Melton models, for example, extend the classic relational databases model with a similarity or proximity relation to obtain a database able of handling inaccurate information [20].

Since the Bousi∼Prolog language uses a proximity or similarity based resolution mechanism, it is suitable for the construction of flexible databases following the aforementioned Buckles-Petry and Shenoi-Melton models. In addition, being a logic programming language that allows to specify facts and rules, Bousi∼Prolog is also useful to build *deductive databases*, that is, databases that have a certain capacity for deduction.

In the Code listing 5.3, we present a simple flexible deductive database extracted from [20] which is implemented in the Bousi∼Prologlanguage.

The database contains information about several films and the theaters where they are projected. It is composed of three tables, represented by facts in the BPL code: `film`, which stores the name, the director and the genre of each film; `theater`, where the name, owner and location of the theaters are stored; and `engagement`, that indicates in which theater(s) each movie can be viewed.

Following the model of Shenoi-Melton, in this database the vagueness is modeled by a proximity relation. In that relation, it is indicated the distances between the different locations of the cinemas and the proximity between the genres of films.

Once the database is loaded into the BPL system, the `search` predicate can be used to search for those theaters near a certain location where movies of the preferred genre (or a similar one) are projected. However, an unrestricted search would return a large number of results, many with a low degree of approximation and, therefore, little relevance. For this reason, before performing the search, it would be convenient to modify the value of the *lambda cut* and assign an intermediate value between 0 and 1, depending on how much you

```
% film(?Title, ?Director, ?Genre)
%   Stores the film table.

film(modern_times, chaplin, comedy).
film(psycho, hitchcock, suspense).
film(robbery, yates, suspense).
film(star_wars, lucas, adventure).
film(surf_party, dexter, drama).


% theater(?Name, ?Owner, ?Location)
%   Stores the theater table.

theater(chinese, mann, hollywood).
theater(odeon, cineplex, santa_monica).
theater(rialto, independent, downtown).
theater(village, mann, westwood).


% engagement(?Film, ?Theater)
%   Stores the engagement table (where is each film displayed on).

engagement(modern_times, rialto).
engagement(star_wars, rialto).
engagement(star_wars, chinese).
engagement(surf_party, village).
engagement(robbery, odeon).
engagement(modern_times, odeon).


% search(+Genre, +Location, -Film, -Theater)
%   Returns a Film which the given Genre which is shown on a
%   Theater close to the specified Location.

search(Genre, Location, Film, Theater) :-
  film(Film, _Director, Genre),
  engagement(Film, Theater),
  theater(Theater, _Owner, Location).


% Proximity relation for distance relation
bervely_hills ~ santa_monica = 0.45.
bervely_hills ~ hollywood = 0.56.
bervely_hills ~ westwood = 0.9.
downtown ~ hollywood = 0.45.
downtown ~ santa_monica = 0.23.
hollywood ~ santa_monica = 0.3.
hollywood ~ westwood = 0.45.
santa_monica ~ westwood = 0.9.

% Proximity relation for genre relation
comedy ~ drama = 0.6.
comedy ~ adventure = 0.3.
drama ~ adventure = 0.6.
drama ~ suspense = 0.6.
adventure ~ suspense = 0.9.
```

Code listing 5.3: A films and theaters deductive database implemented in bpl [20].

want to restrict the search.

For example, if you set the value of the *lambda cut* to 0.4 and launch the goal

```
search(adventure, downtown, Film, Theater)
```

to search the movies of adventures near `downtown`, the result will be this:

```
BPL> lc 0.4
New lambda-cut value is:  0.4
BPL> search(adventure, downtown, Film, Theater)
Film = star_wars
Theater = rialto
With approximation degree:  1 ;
Film = star_wars
Theater = chinese
With approximation degree:  0.45
```

With the information stored in the database, the system has determined that there are two movies and theaters that meet the conditions indicated in the query. For example, one of the solutions obtained is that in the cinema `chinese`, despite not being located in `downtown` , you can see the movie `star_wars`; this result has an approximation degree of 0.45 because the theater is located in `hollywood`, and the locations `hollywood` and `downtown` are related with degree 0.45.

5.3. APPROXIMATE REASONING

Approximate reasoning is, essentially, the inference of an imprecise conclusion from imprecise premises [38]. This type of reasoning is common in artificial intelligence applications because it is more like the way people think and act than the exact reasoning, since many real-world concepts are neither absolute truths nor absolute falsehoods.

In a logic program, the approximate reasoning can be carried out through fuzzy inferences, understanding a fuzzy inference as the application of a generalized *modus ponens* inference rule in which the known fact does not coincide exactly with the one that appears in the antecedent (condition) of the conditional sentence.

Thus, while in traditional logic the *modus ponens* rule is defined as follows:

If A is true, then B is true

A is true

B is true

The generalized *modus ponens* rule of fuzzy logic could be expressed intuitively as follows [45]:

If A is true, then B is true

A' (which is close to A) is true

B' (which is close to B) is true

To see how the Bousi∼Prolog language can be useful in modeling approximate reasoning, we will start with the following fuzzy inference, taken from [38]:

If X is *young*, then X is *fast*

Bill is *middle* aged

Bill is *somewhat fast*

The Code listing 5.4 shows a possible Bousi∼Prolog formalization of the above inference. To represent the concepts «young» and «fast» we have chosen to define two linguistic variables, *age* and *speed*, containing these two linguistic terms.

When the program of Code listing 5.4 is loaded into the BPL system, internally, a fuzzy proximity relation is generated that relates the linguistic terms of each linguistic variable. This relation is what enables the system to produce the following approximate answer when it is asked if Bill is fast.

```
BPL> speed(bill, fast)

Yes
With approximation degree:  0.325
```

In the example program that has just been explained, the age of three other people has also been formalized. For this, the fuzzy subset construction operator `#` has been used. As you can see, with this operator it is possible to indicate in a very intuitive way people ages, either in an exact or approximate way. In the example, Lisa is 20 years old, Robert is only known to be very old and Susan is between 30 and 40 years old.

With these new three facts, it would be interesting to ask the BPL system to compute which people are fast. The answer we would get in that case would be the following:

```
% Linguistic variable 'age'
:- domain(age, 0, 100, years).
:- fuzzy_set(age, [baby(0,0,5),
                   young(0,10,30,40),
                   middle(20,40,60,80),
                   old(50,80,100,100)]).

% Linguistic variable 'speed'
:- domain(speed, 0, 40, kmh).
:- fuzzy_set(speed, [slow(0,0,10,20),
                     normal(15,20,25,35),
                     fast(25,30,40,40)]).

% Rules
speed(Person, fast) :- age(Person, young).

% Facts
age(bill, middle).

% Additional facts
age(lisa, age#20).
age(robert, very#old).
age(susan, about#age#30#40).
```

Code listing 5.4: An example of approximate reasoning in Bousi∼Prolog [38].

```
BPL> speed(Person, fast)

Person = bill
With approximation degree:  0.325 ;

Person = lisa
With approximation degree:  1 ;

Person = susan
With approximation degree:  0.5
```

According to the BPL system, for the four people that appear in the program, the only one which is certainly not fast is Robert. This fact makes sense because Robert's age is determined by the linguistic term «very old» and this is not close to «young».

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

In this chapter a summary of the most outstanding characteristics of the developed system is made and its advantages, limitations and some conclusions are commented. It also includes several proposals with which the system and our work could be continued, improved and expanded.

6.1. RESUMEN

We have designed A fuzzy logic programming language called Bousi∼Prolog, and the necessary tools to create, edit and execute programs written in this language have been implemented.

The Bousi∼Prolog language is an extension of Prolog that uses a variant of the SLD resolution strategy called WSLD resolution. This new resolution strategy replaces the classical unification algorithm of the SLD resolution strategy with the weak unification algorithm proposed in [39]. Thanks to it, the language can intuitively handle vagueness, while at the same time makes the query answering process more flexible.

Through the operators `~`, `~>`, `<~`, `~1~`, `~2~` and `~3~` up to six fuzzy relations with different properties of reflexivity, symmetry and transitivity can be defined. The relation ∼ intervenes in the core of the weak unification algorithm and, therefore, in the WSLD resolution process. This is the one that allows the management of vague information and the flexible answer to the queries, while the others can be used to build synonyms and hierarchies or, in general, arbitrary relationships and make queries about them.

Also, the Bousi∼Prolog language facilitates the definition of linguistic terms and their association with fuzzy subsets thanks to the directives `domain` and `fuzzy_set`. Moreover, it allows the construction of composite linguistic terms with the operator `#`. Following the proposals of [23, 19], the information represented by the the fuzzy subsets (associated to the linguistic terms) becomes a fuzzy binary relation on the set of linguistic terms, so that the system can compare the proximity between these terms using the same methods as in the rest of the fuzzy relations.

The system that implements the Bousi∼Prolog language has been called BPL system and it has been essentially divided into three subsystems: the *command processor*, which deals with the input commands entered by the user; the *compiler*, whose function is to translate the Bousi∼Prolog programs into an intermediate representation; and the *interpreter*, in charge of executing the queries and computing their answers.

The approach that we have followed to implement the WSLD resolution strategy, which is the operational mechanism of the Bousi∼Prolog language, consisted on translating the BPL code of the programs into a code that has been called translated BPL code or TPL code[1]. The TPL files storing that code only have Prolog sentences, but they explicitly implement the computation of approximation degrees and weak unification thanks to the incorporation of additional parameters and calls to auxiliary predicates implemented by the interpreter.

In the development of the BPL system we have followed methodologies according to the technologies and the type of system built, and we have been carried out the five classic phases of Software Engineering: requirement specification, analysis, design, implementation and testing.

6.2. ADVANTAGES AND LIMITATIONS

Compared to other existing fuzzy logic programming systems, one of the main advantages of the BPL system is that it is based on a powerful Prolog implementation and in accordance with the ISO standard, such as SWI-Prolog. In this way, the BPL system inherits a extensive predicate library from SWI-Prolog and, thanks to a specific translation process, supports special operators, some higher-order predicates, the cut and a indexing mechanism for clauses.

Another contribution of the developed system is its support for the definition of both similarity and proximity relations, which allows the use of this language in situations where the transitive property of a fuzzy relation is not desirable. However, in specific cases the use of proximity relations in combination with the algorithms that are currently implemented can lead to incompleteness problems (that is, some answers may be lost), although always maintaining the correction (that is, the answers obtained are always correct). These problems and their possible solutions are discussed in [38].

Also noteworthy is the integration in Bousi∼Prolog language of the handling of linguistic variables and fuzzy subsets, and the wide variety of mechanisms that are provided to

[1]Compare this approach with the one followed in [16, 17, 19], where an extension of the Warren abstract machine is used to execute the BPL code.

represent domain points, domain ranges and modifiers of other fuzzy subsets using a simple, intuitive syntax.

One of the major practical limitations of the BPL system is the one related with the efficient consumption of time and memory in the execution of queries. Although this consumption is notably lesser than the consumption experimented by the first prototype of the BPL system (which was a metainterpreter), a drop in performance is still observed when executing programs of medium or high complexity.

The BPL system also presents some limitations to handle errors of source code files, given the large number of error situations that may occur. Nevertheless, the most common mistakes made by Prolog programmers have been taken into account, leaving aside a more specialized treatment as a future improvement.

6.3. FINAL CONCLUSIONS

The work done in the present work is an example of how engineering in general and computer science in particular can be used to put into practice ideas and scientific theories, in order to obtain a feedback that allows to deepen and improve those theories.

The main fruit of this project, the BPL system, covers the fields of logic programming, fuzzy logic and the theory of fuzzy sets, and can be classified within the framework of the so-called similarity-based fuzzy logic programming systems.

This project shows that the different programming paradigms that currently exist do not have to be used independently, but can be combined to take advantage of the capacities of each one of them. Not in vain, in the implementation of the BPL system we have used two programming languages (Prolog, and C) belonging to different paradigms (declarative and structured).

As a final conclusion, it is important to say that when this project was started, one of the main objectives was to make it easier for regular Prolog programmers, and also programmers of other programming languages, to have access to the Bousi∼Prologlanguage on which the BPL systemis based. This objective has been reached for four reasons:

- The Bousi∼Prolog language is an extension of Prolog, not a completely new language. Except for certain specific sentences, most of its syntax will be familiar to Prolog programmers. However, as it was said, it embodies a richer operational semantics which transparent to the user of the system allowing him/her a greater expressivity.

- The BPL system is a command-line application with a similar structure to the current Prolog interpreters. The execution of queries and the presentation of results is done in the same way as in SWI-Prolog, GNU Prolog or YAP.
- The BPL system is publicly available on the website of the research group Dec-tau[2] and can be installed on Windows, Linux and Mac OS X. Also, the BPL system can be run on-line through of a tool which is accessible in[3].

6.4. FUTURE WORK

Bousi∼Prolog is a programming language of recent creation and, as such, there are many aspects that can be improved to give the language more expressiveness and versatility. Some of the most interesting future work lines for the BPL system are those mentioned below:

- **To assign weight annotations to the program clauses**. This characteristic, derived from fuzzy logic programming languages that use graded rules and partially implemented in systems such as Prolog-ELF [13] or FProlog [29], consists of assigning truth degrees (or some kind of weights) to the clauses that make up a program. Its inclusion in the Bousi∼Prolog language would allow to solve some limitations that proximity-based fuzzy logic programming languages present.
- **Improvements handling fuzzy sets**. To adapt the use of the BPL system to control applications, it would be convenient to implement *defuzzification* mechanisms that would allow the results of inferences with fuzzy subsets to be converted into crisp values, something that has already does on systems like FuzzyCLIPS [33] or (partially) the low level implementation of the BPL system [19].
- **Maintenance of ontology repositories**. For the design of systems involving large volumes of data it would be useful to create repositories of ontologies, that is, collections of ontologies that could be loaded and downloaded at runtime.
- **Debugging**. The existence of a debugger is essential so that a programming language can be considered as a real alternative for the development of medium-sized systems.

[2]https://dectau.uclm.es/bousi-prolog
[3]https://dectau.uclm.es/BPLweb

A future line of work of the BPL system would be the creation of a debugger with which the execution of the BPL code could be followed[4].

[4]It is now possible to use the underlying SWI-Prolog debugger to execute programs written in the Bousi∼Prologlanguage step by step, but the code that is actually debugged is the TPL code.

ANEXO A. INSTALLATION MANUAL OF THE BPL SYSTEM

This appendix explains the steps that you must follow to install and compile the source code of the BPL system developed in this work.

A.1. LINUX AND MAC OS X

The Linux version of the BPL system requires to have installed the packages of the following programs and libraries in order to compile correctly its source code:

- SWI-Prolog 5.7 or higher, including the development package (if any). If SWI-Prolog is not part of the Linux distribution that is being used, alternatively you can manually download and compile its source code [1].
- Editline Library 2.11 or higher, including the development package.
- Flex 2.5.4 or higher.
- GCC 4.0 or higher.
- GNU Make 3.81 or higher.

On the other hand, to be able to compile and execute the BPL system in the Mac OS X operating system it is essential to have previously installed these applications:

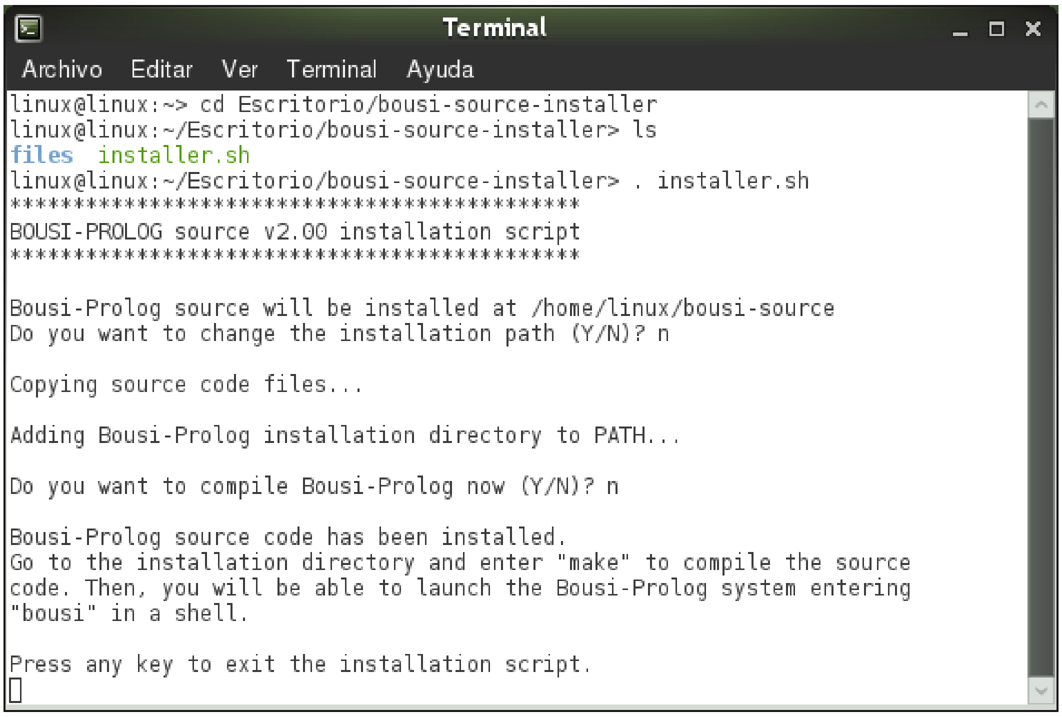
- SWI-Prolog 5.7 or higher.
- Xcode Tools 3.2 or higher[2]. From this package, at least the development tools for UNIX must be installed.

Once the necessary programs and development libraries have been installed, the steps that must be followed to copy, configure and compile the BPL system in the Linux or Mac OS X operating systems are the following.

1. Unzip the package *bousi-source-linux-installer.tar.gz* or *bousi-source-macosx-installer.tar.gz*(dependi on the platform) found on the Bousi∼Prolog website in any temporary directory of the hard disk.

```
$ cd <temp_directory>
$ tar xzf <ruta_DVD>/bousi-source-<your_system>-installer.tar.gz
```

[1] http://www.swi-prolog.org/
[2] http://developer.apple.com/technologies/tools/



```
linux@linux:~> cd Escritorio/bousi-source-installer
linux@linux:~/Escritorio/bousi-source-installer> ls
files  installer.sh
linux@linux:~/Escritorio/bousi-source-installer> . installer.sh
*********************************************
BOUSI-PROLOG source v2.00 installation script
*********************************************

Bousi-Prolog source will be installed at /home/linux/bousi-source
Do you want to change the installation path (Y/N)? n

Copying source code files...

Adding Bousi-Prolog installation directory to PATH...

Do you want to compile Bousi-Prolog now (Y/N)? n

Bousi-Prolog source code has been installed.
Go to the installation directory and enter "make" to compile the source
code. Then, you will be able to launch the Bousi-Prolog system entering
"bousi" in a shell.

Press any key to exit the installation script.
```

Figure A.1: Installation *Script* of the BPL system in a Linux operating system.

2. Execute in a terminal the *installer.sh script* that will be created inside the directory *bousi-source-installer* and follow the instructions that appear on the screen (see Figure A.1). The installation wizard will ask for the directory in which you want to copy the source code and will ask if you want to compile it now or later.

```
$ cd <temp_directory>/bousi-source-installer
$ .  installer.sh
```

By default, the BPL system is installed in a local directory of the active user, but if the *installer.sh script* is run as administrator you will have the option of installing the BPL system for all users, so that after the program is compiled, any user can execute it.

3. When the installation is complete, you can delete the directory *bousi-source-installer* created in step 1.

```
$ rm -r <temp_directory>/bousi-source-installer
```

4. If the source code of the BPL system was not compiled during the installation, it can be done at any time from a terminal by going to the folder where the application was stored and running the tool *make* without arguments.

```
$ cd <installation_directory>
$ make
```

5. Once the BPL system has been compiled, to start it you just have to execute the command *bousi* in a terminal from any directory.

```
$ bousi
```

If the BPL system was installed as an administrator and for all users, any other user of the system can also launch it by typing *bousi* in a terminal.

6. Optionally, in addition to the executable image of the BPL system, it is also possible to generate the test bench and the code documentation as explained below:

- To compile the test bench, the command *make bousitest* must be invoked from the installation folder. After the compilation, the program *bousitest* can be run to launch all the tests and see the coverage reached by them.

  ```
  $ cd <installation_directory>
  $ make bousitest
  $ ./bousitest
  ```

- The documentation of the source code can be generated automatically with the command *make doc.pdf*. It must be taken into account that a distribution of LaTeX must be installed in the system to generate the documentation in PDF format.

  ```
  $ cd <installation_directory>
  $ make doc.pdf
  ```

7. When you want to uninstall the BPL system you have to execute the *uninstaller.sh script* that will be in the folder where the source code of the BPL system was copied. The uninstall wizard will ask for confirmation before making any changes.

```
$ cd <installation_directory>
$ ./uninstaller.sh
```

A.2. WINDOWS

En Windows es necesario tener instaladas las siguientes aplicaciones para poder compilar y ejecutar correctamente el BPL system: In the Windows operating system it is necessary to have installed the following applications in order to compile and correctly execute the BPL system:

- SWI-Prolog 5.7 or higher[3].
- Flex 2.5.4 or higher[4]. It is recommended to install the complete package that does not include the source code.
- Windows SDK 6.0 or higher[5]. From this development kit you should at least install

[3] http://www.swi-prolog.org/
[4] http://gnuwin32.sourceforge.net/packages/flex.htm
[5] http://msdn.microsoft.com/es-es/windows/bb980924

the Visual C ++ compilers, the header files and the libraries for the x86 platform. If Visual Studio 2005 SP1 or some later version is already installed in the system, instead of installing the Windows SDK separately, support for Visual Studio Visual C ++ programming should be installed.

The following describes step by step how to install and compile the source code of the BPL system under the Windows operating system.

1. Before proceeding with the installation, the following modifications must be made to the system environment variables:

   1.1. Add to the environment variable *PATH* the path to the directory where the file *vcvars32.bat* of the Windows SDK is located. This path will depend on the operating system that is being used and the version of the Windows SDK installed. For example, for Windows 7 and Windows SDK 7.0, the appropriate path would be «*C:\Program Files\Microsoft Visual Studio 9.0\VC\bin*».

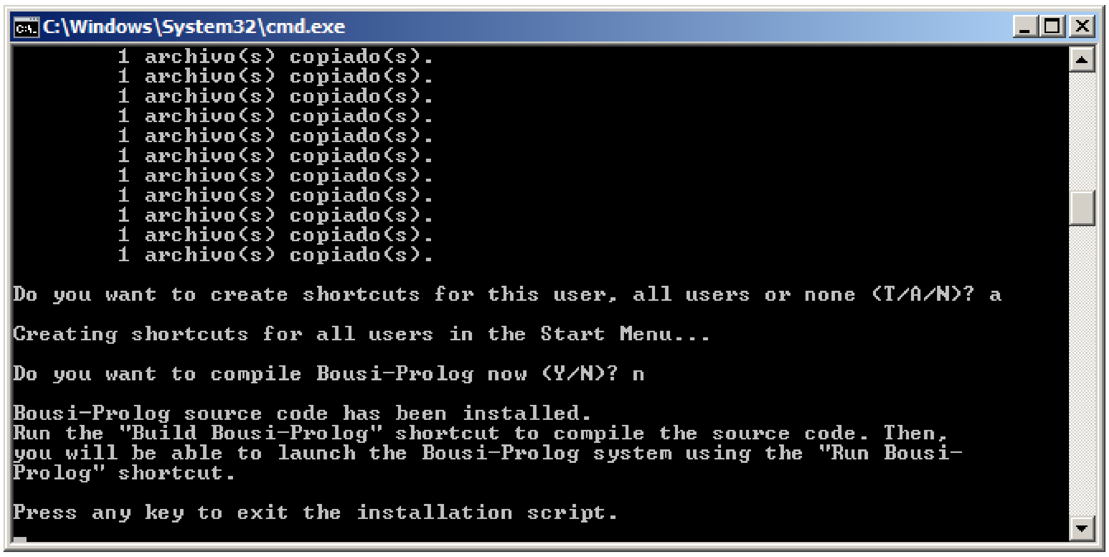
   1.2. Also add to the variable *PATH* the path of the subfolder *bin* of the directory where SWI-Prolog was installed. This path should be similar to «*C:\Program Files\pl\bin*».

   1.3. Create a new environment variable called *GNUWIN32* that points to the base directory in which the Flex tool is installed. Typically the path of this folder will be similar to «*C:\Program Files\GnuWin32*».

2. Unzip the package *bousi-source-windows-installer.zip* found on the Bousi∼Prolog website in any temporary directory of the hard disk.

3. Open in a Windows Explorer window the directory *bousi-source-installer* that will have been created in the previous step, run the file batch *installer.bat* and follow the instructions on the screen (see Figure A.2). The installation wizard will ask for the path where you want to copy the source code and also will ask if you want to compile it now or later and if you want to create shortcuts in the Init Menu (see Figure A.3).
In Windows Vista and Windows 7 all batch files must be run as administrator if you intend to install the BPL system in a protected directory of the system (such as «*C:\Program Files*»).

4. When the installation is complete, you can delete the *bousi-source-installer* directory created in step 2.



```
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).
        1 archivo(s) copiado(s).

Do you want to create shortcuts for this user, all users or none (T/A/N)? a

Creating shortcuts for all users in the Start Menu...

Do you want to compile Bousi-Prolog now (Y/N)? n

Bousi-Prolog source code has been installed.
Run the "Build Bousi-Prolog" shortcut to compile the source code. Then,
you will be able to launch the Bousi-Prolog system using the "Run Bousi-
Prolog" shortcut.

Press any key to exit the installation script.
_
```

Figure A.2: Batch file of installation of the BPL system in Windows.

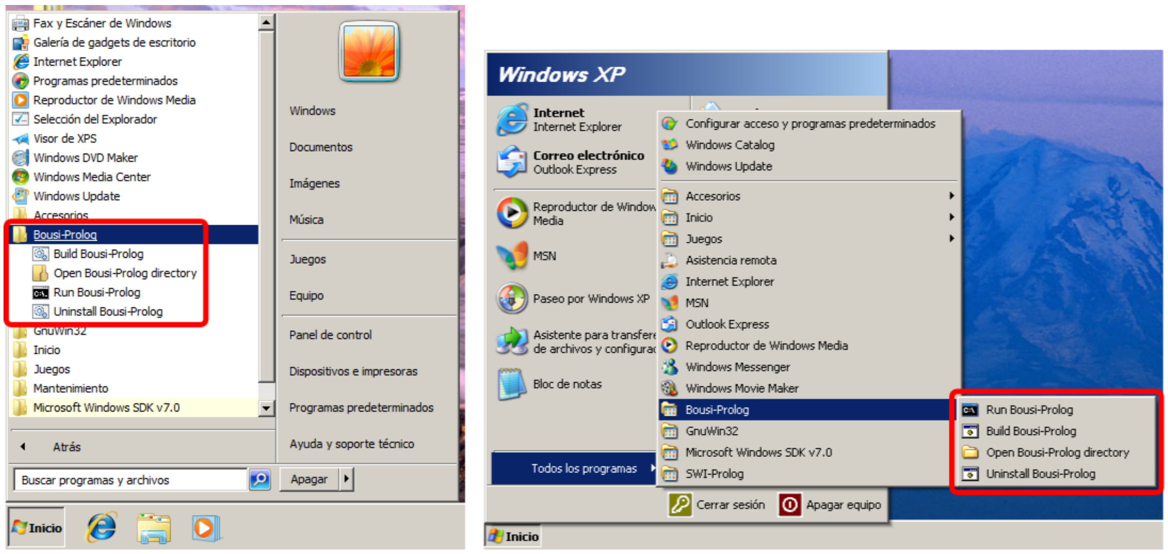


Figure A.3: Shortcuts created by the installer of the BPL system in Windows.

5. If the source code of the BPL system was not compiled during the installation process, it can be done at any time by launching the shortcut *Bousi Prolog* → *Build Bousi-Prolog* from the Init Menu or by opening the folder where it was installed the BPL system and running the file batch *make.bat*.

6. Once the BPL system has been compiled, to start it, you must use the shortcut *Bousi Prolog* → *Run Bousi-Prolog* from the Init Menu, or you can open the instalation directory and execute the file *bousi.exe*.

7. Optionally, in addition to the executable image of the BPL system, the test bench and the code documentation can also be generated as indicated below:

   - To compile the test bench you have to execute the batch file *maketest.bat* located in the installation folder. Then, with the program *bousitest.exe* you can launch all the tests and see the coverage achieved by them.

- The documentation of the source code can be generated by launching the batch file *makedoc.bat*. The documentation will be stored in the file *doc.pdf*, provided that a distribution of LaTeX has been installed in the system that allows generating the documentation in PDF format.

8. When you want to uninstall the BPL system, you must use the shortcut *Bousi Prolog* → *Uninstall Bousi-Prolog* from the Init Menu or open the installation folder and run the batch file *uninstaller.bat*. The uninstall wizard will ask for confirmation before making any changes.

ANEXO B. BPL SYSTEM USER MANUAL

This appendix contains the user manuals of the BPL system developed in this work.

The BPL system is a command-line interpreter that allows you to load and execute programs written in the Bousi∼Prolog language, an extension of the Prolog logic programming language with fuzzy logic features. The main characteristics and syntax of Bousi∼Prolog are fully explained in Chapter 3 and will not be repeated in this manual. Therefore, we focus our attention on the BPL shell.

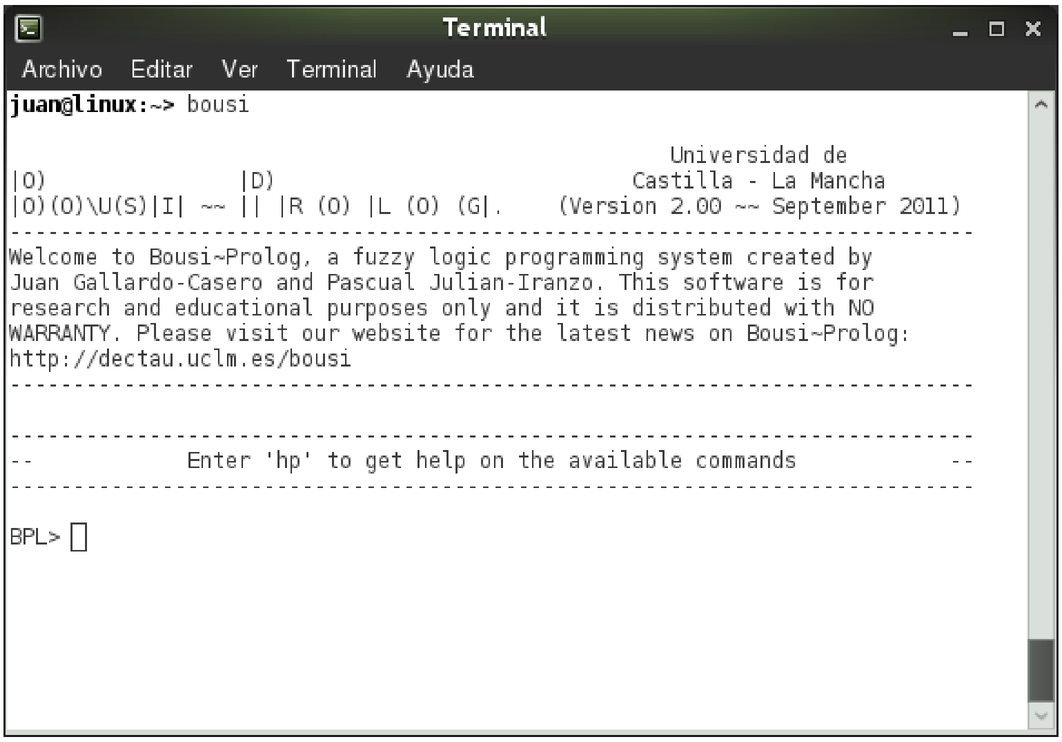
As can be seen in Figure B.1, when the BPL system starts, a welcome message is displayed, after which the *prompt* `BPL>` appears, indicating that the system is waiting to receive a command from the user.

The BPL system has a total of nine commands, which are summarized below:

- `ld`. Loads a BPL source file containing a program or ontology, or show the name of the files currently loaded in memory.
- `sv`. Executes a query (you can also enter queries without using `sv`).
- `lc`. Shows or sets the minimum approximation degree (known as *lambda cut*) allowed in a weak unification process.
- `pwd`. Shows the path of the current working directory.
- `cd`. Changes the current working directory.
- `ls`. Displays the contents of the current working directory.
- `hp`. Offers general help on the BPL system or information about a command.
- `sh`. Launches a new *shell* or command line interpreter without leaving the BPL system.
- `qt`. Quits the BPL system

Each command must be entered on a separate line and, unlike what happens in Prolog, it is not needed to add a period to the end of any command. In commands that can receive a list of options (such as `ld`), this list should always be the first parameter.

Although the command line functionalities depend on the operating system being used, all versions of the BPL system have a history that stores the last 100 commands entered. Through the ↑ and ↓ keys you can navigate through this history to repeat or modify any of the previously released orders.



```
juan@linux:~> bousi

                                                    Universidad de
|O)                 |D)                           Castilla - La Mancha
|O)(O)\U(S)|I| ~~ || |R (O) |L (O) (G|.     (Version 2.00 ~~ September 2011)
---------------------------------------------------------------------------
Welcome to Bousi~Prolog, a fuzzy logic programming system created by
Juan Gallardo-Casero and Pascual Julian-Iranzo. This software is for
research and educational purposes only and it is distributed with NO
WARRANTY. Please visit our website for the latest news on Bousi~Prolog:
http://dectau.uclm.es/bousi
---------------------------------------------------------------------------

---------------------------------------------------------------------------
--            Enter 'hp' to get help on the available commands           --
---------------------------------------------------------------------------

BPL> 
```

Figure B.1: Wellcome to the BPL system.

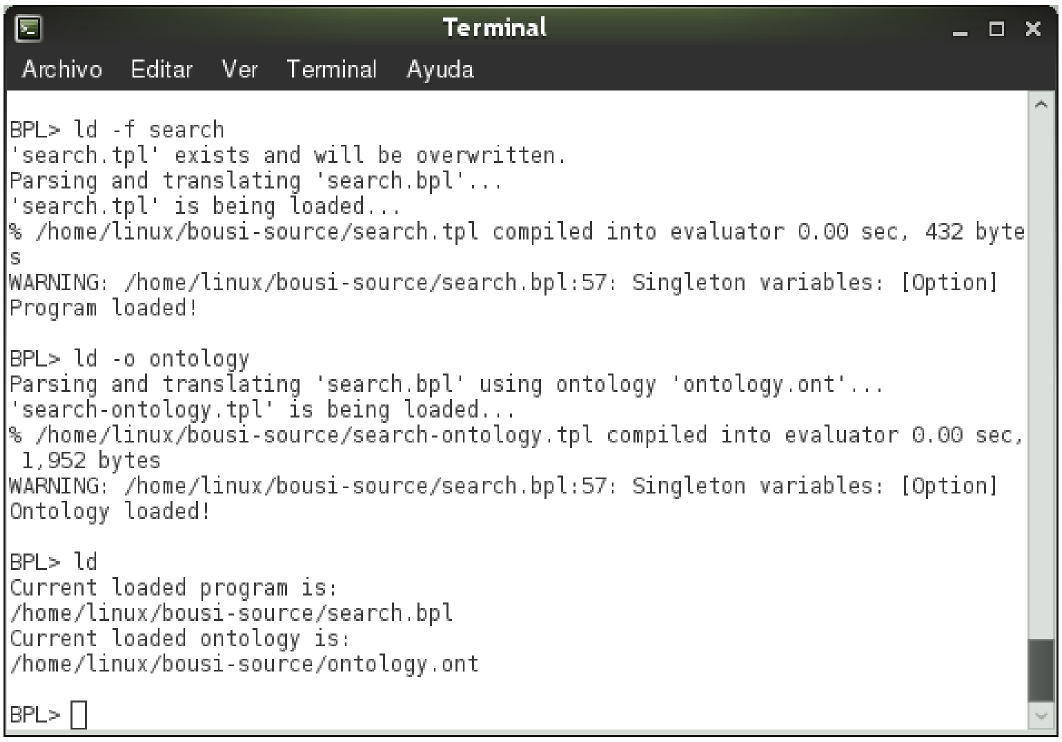
In the following pages, the utility and the specific syntax of each command are explained in more detail.

Loading programs (`ld` command)

The `ld` command reads, compiles and loads a source code file written in the Bousi∼Prolog language, which can contain both a program and an ontology. In the BPL system, a file that only contains proximity relations and / or directives (that is, it has no facts or rules) is called an *ontology*.

The BPL system can keep a program and an ontology in memory simultaneously, so that the same program can be executed with different ontologies in a simple way. When a program is loaded and then an ontology, the equations in both code files are combined; if a new ontology is subsequently loaded on the same program, only the equations defined in the previous ontology are downloaded, maintaining both the rules and the equations that the program had.

The default extensions used by the BPL system are *.bpl* for programs and *.ont* for ontologies. When the program or ontology that you want to load has the default extension, it is not necessary to indicate it in the load command.



```
BPL> ld -f search
'search.tpl' exists and will be overwritten.
Parsing and translating 'search.bpl'...
'search.tpl' is being loaded...
% /home/linux/bousi-source/search.tpl compiled into evaluator 0.00 sec, 432 byte
s
WARNING: /home/linux/bousi-source/search.bpl:57: Singleton variables: [Option]
Program loaded!

BPL> ld -o ontology
Parsing and translating 'search.bpl' using ontology 'ontology.ont'...
'search-ontology.tpl' is being loaded...
% /home/linux/bousi-source/search-ontology.tpl compiled into evaluator 0.00 sec,
 1,952 bytes
WARNING: /home/linux/bousi-source/search.bpl:57: Singleton variables: [Option]
Ontology loaded!

BPL> ld
Current loaded program is:
/home/linux/bousi-source/search.bpl
Current loaded ontology is:
/home/linux/bousi-source/ontology.ont

BPL> 
```

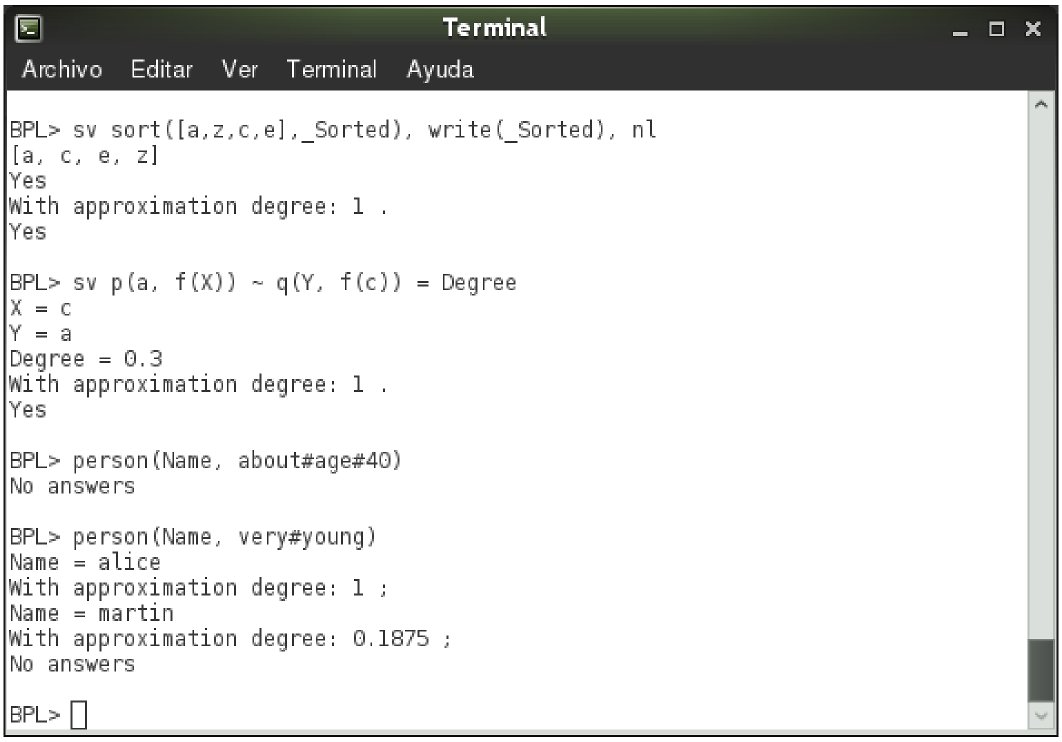
Figure B.2: Loading a program and an ontology in the BPL system.

All the source code files Bousi∼Prolog are translated into an intermediate representation in Prolog that is called TPL code. This code is stored in files with the same name as their corresponding BPL files but with the extension *.tpl*. To reduce the load time, when trying to load a BPL file that has not been modified since it was compiled to TPL for the last time, the system loads its TPL code directly without reading or translating the BPL code. To ignore the contents of the TPL file and force the recompilation of the source code, you can pass the `-f` option.

On the other hand, when used without arguments, the command `ld` shows the path of the program and the ontology that are currently loaded in memory. In Figure B.2 you can see this and the two previous uses of the `ld` command.

Sintaxis:

♦ `ld`	shows the path of the program and the loaded ontology
♦ `ld <file>`	translates and loads the BPL program contained in `<file>`
♦ `ld -f <file>`	forces the compilation of the BPL `<file>` and then loads it
♦ `ld -o <file>`	translates and loads the BPL ontology contained in `<file>`
♦ `ld -fo <file>`	forces the compilation of the BPL ontology `<file>` and then loads it



```
BPL> sv sort([a,z,c,e],_Sorted), write(_Sorted), nl
[a, c, e, z]
Yes
With approximation degree: 1 .
Yes

BPL> sv p(a, f(X)) ~ q(Y, f(c)) = Degree
X = c
Y = a
Degree = 0.3
With approximation degree: 1 .
Yes

BPL> person(Name, about#age#40)
No answers

BPL> person(Name, very#young)
Name = alice
With approximation degree: 1 ;
Name = martin
With approximation degree: 0.1875 ;
No answers

BPL> 
```

Figure B.3: Execution of several queries in the BPL system.

Running queries (`sv` command)

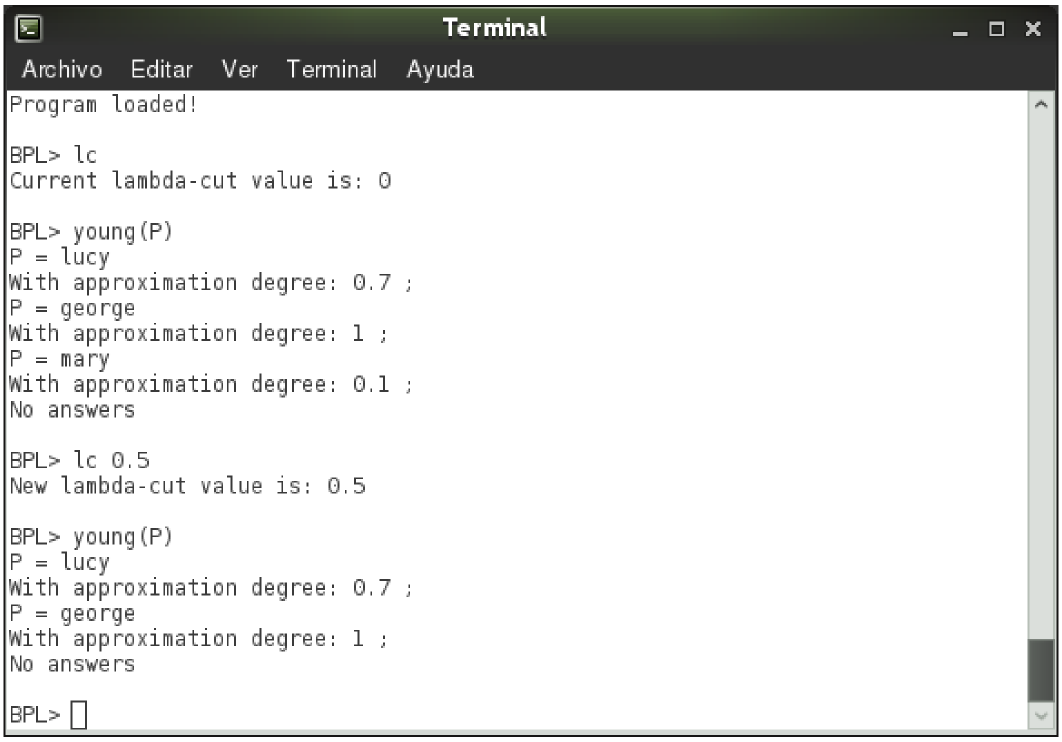
The `sv` command executes a query (*query*) about the program (and, possibly, the ontology) currently loaded into memory using the WSLD resolution operational mechanism. You can also launch queries when there is not a program loaded in memory, although in that case only the predefined predicates of the system will be available.

In a Bousi∼Prolog query any statement or expression that is valid in the body of a clause can appear. This includes both simple and compound Prolog terms as well as Bousi∼Prolog specific expression like term comparisons and weak unification of terms.

When the BPL system finds at least one solution for the query passed as a parameter, its approximation degree and the binding of the variables that appeared in the query are shown. Then you can type a semicolon (`;`) to find more solutions, or press *Enter* to end the query. As soon as the BPL system does not find more solutions, or if the query has no solution, the message *No answers* will be displayed on the screen. The Figure B.3 shows several query examples in which these situations occur.

The `sv` command is the default system command, which means that you can also launch queries without needing to be preceded by `sv`.

Sintaxis:



```
Program loaded!

BPL> lc
Current lambda-cut value is: 0

BPL> young(P)
P = lucy
With approximation degree: 0.7 ;
P = george
With approximation degree: 1 ;
P = mary
With approximation degree: 0.1 ;
No answers

BPL> lc 0.5
New lambda-cut value is: 0.5

BPL> young(P)
P = lucy
With approximation degree: 0.7 ;
P = george
With approximation degree: 1 ;
No answers

BPL> 
```

Figure B.4: Querying and modification of the *lambda cut* in the BPL system.

♦ `sv` *<query>* .... executes the indicated *<query>* using WSLD resolution

♦ *<query>* .... same as `sv` *<query>*

Management of the *lambda cut* (`lc` command)

Using the `lc` command, you can consult or modify the minimum approximation degree allowed for a weak unification process to be successful. This minimum approximation degree is called *lambda cut* in Bousi∼Prolog. The default *lambda cut* of a program is 0 unless it contains a directive `lambda_cut`; however, when `lc` is used to set a new *lambda cut* the value indicated in the aforementioned directive is overwritten.

The main utility of the *lambda cut* is to limit the expansion of the WSLD resolution search tree. When the *lambda cut* is set to a value greater than 0, the weak unification process will fail each time the resultant approximation degree s less than the *lambda cut*, so the resolution step where the unification was executed will also fail and it will stop exploring a branch of the search tree that with the default *lambda cut* would have been analyzed. Figure B.4 presents a query example that produces different results depending on the value of the *lambda cut*.

Sintaxis:

- `lc` .... shows the current value of the *lambda cut*
- `lc` *`<degree>`* .... sets *`<degree>`* as the new value of the *lambda cut*, where *`<degree>`* is a real number between 0 and 1 (both inclusive)

Getting help (`hp` command)

The `hp` command allows you to obtain help on the available commands of the BPL shell. With this help command you can either consult the complete list of supported commands as well as show the description and syntax of some of them.

Sintaxis:

- `hp` .... shows the list of available commands in the BPL system
- `hp` *`<command>`* ...... shows the description and syntax of the indicated *`<command>`*

Querying the contents of the working directory (`ls` command)

As in a Unix terminal, the `ls` command displays the files and folders in the current working directory, excluding those files and folders whose names begin with a period (`.`).

Sintaxis:

- `ls` .... shows the contents of the current working directory

Querying of the working directory (`pwd`)

The `pwd` command is used to check the absolute path of the current working directory.

Sintaxis:

- `pwd` .... shows the complete path of the current working directory

Modification of the working directory (`cd` command)

With the `cd` command you can change the working directory. Paths that include names of directories with spaces must be enclosed in quotation marks so that the system interprets them correctly.

Sintaxis:

- `cd` *`<path>`* .... sets *`<path>`* as the new working directory; *`<path>`* can be both an absolute or relative path

Executing a *shell* (`sh` command)

The `sh` command allows you to run a new *shell* or command line without leaving the BPL system. By default, the *shell* that is thrown when this command is executed is */bin/sh* (in Linux and Mac OS X) or *cmd.exe* (in Windows), but it can be execute a different command interpreter by entering its path in the environment variables *SHELL* (in Linux and Mac OS X) or *COMSPEC* (in Windows).

Sintaxis:

♦ `sh` .... launches a new *shell* without abandoning the BPL system

Quitting the BPL system (`qt` command)

The `qt` command exits immediately from the BPL system. You should always use this command instead of the predefined Prolog `halt` predicate when you want to leave the BPL system so that the history is saved correctly.

Sintaxis:

♦ `qt` .... quits the BPL system

ANEXO C. GRAPHIC NOTATION FOR PROLOG PROGRAMS

This appendix describes the symbolic notation for Prolog application design proposed by G. Karam in [24]. This notation is the one that has been adopted for the elaboration of the design diagrams of the BPL system.

C.1. INTRODUCTION

The notation described in [24] and explained in this appendix is a variant of Buhr's notation for the design of concurrent systems with Ada [5]. Both are oriented to the design of modular systems, and use a reduced set of icons with a perfectly defined semantics.

Following the *top-down* approach that is commonly used in the development of declarative programs, the notation of [24] can be used to construct two kinds of design diagrams:

- **External view diagrams**. They show the behavior of the system at a high level. Only the public interfaces of each module and the relationships between them are represented, abstracting from the rest of the details of their implementation.
- **Internal view diagrams**. They show the behavior of a single module at a low level, isolating it from the rest of the modules of the system. The internal elements that implement the functionality of the module and its public interface are represented in greater depth.

In the following sections we proceed to explain in detail the meaning of each of the symbols that can be used in both diagrams.

C.2. EXTERNAL VIEW DIAGRAMS

The external view diagrams show the dependencies between the different modules that make up an application, as well as the flow of the data exchanged between them. In these diagrams the modules are treated as *black boxes*, so that no details of their implementation are represented, except for those predicates that are visible from the outside.

In Figure C.1 there is an external view diagram for an example system consisting of two modules. The elements that can appear in this type of diagram are indicated below.

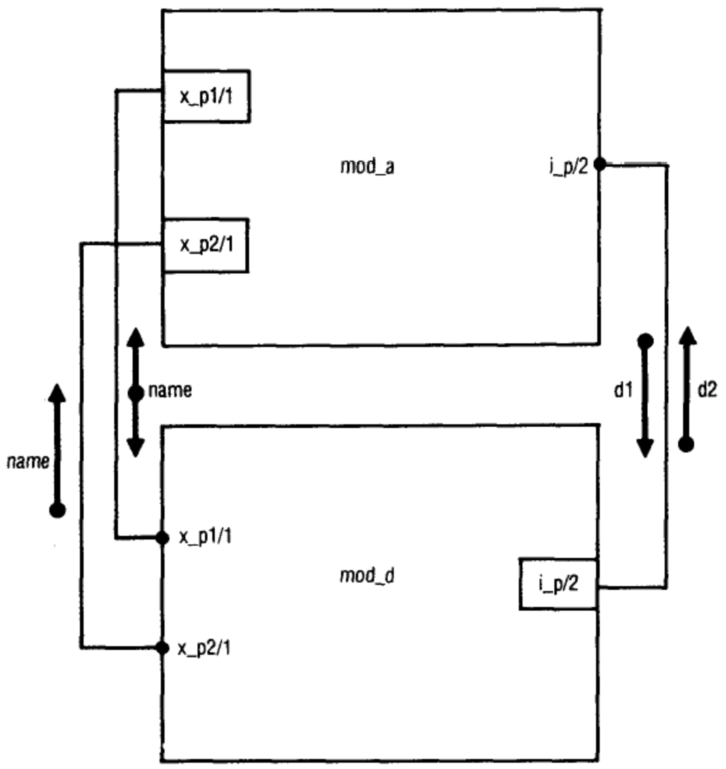
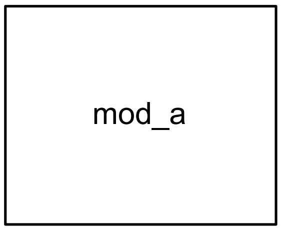


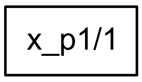
Figure C.1: Example of the external view diagram of a simple system [24].

MODULE



At the highest level of the diagrams are the modules of the system being modeled, represented as rectangles labeled with the name of each module.

EXPORTED PREDICATE



Within a module, a rectangle of this type symbolizes an exported predicate. The set of exported predicates from a module forms the public interface that the rest of the modules can use to access its functionality. Following Prolog standard notation, each predicate is uniquely identified by its functor (name) and arity (number of arguments).

IMPORTED PREDICATE

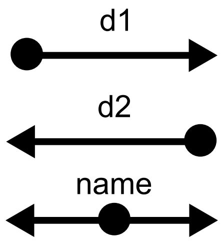
i_p/2 ● A solid circle on the edge of a module represents an imported predicate, that is, a dependency on the use of a predicate exported from another module.

CALL

The lines that connect the different modules of the system denote the calls that the modules make to each other, and allow to see the dependencies between them.

Each line must connect a predicate imported from a module with a predicate exported from another module that has the same signature (functor and arity). The origin of the call is always the imported predicate, while the destination is the exported predicate.

INPUT, OUTPUT OR INPUT/OUTPUT PARAMETER



These arrows symbolize the flow of data that occurs in calls between modules. They can be of three kinds:

- An arrow pointing to the destination of the call indicates that the parameter must be completely instantiated at the time of the call (input parameter).
- If the arrow points to the origin, the argument must never be instantiated, but must be a free variable (output parameter).
- An arrow with two points denotes a parameter that may or may not be instantiated in the call, or a parameter that must be partially instantiated[1] (input/output parameter).

C.3. INTERNAL VIEW DIAGRAMS

The internal view diagrams are intended to show the internal structure of a particular module and the interaction between its most important components (predicates). They intend to serve as the basis for the implementation of the modules, but without falling into an excessive level of detail or incorporating specific characteristics of any specific programming language.

It is important to emphasize that the level of detail of this class of diagrams will depend to a great extent on the phase in which the design is located. In fact, design diagrams must be refined as the development and understanding of the system progresses.

[1]A partially instantiated parameter is one that has a fixed structure but is made up of non-instantiated variables. A typical case is a list in which both the head and the tail are free variables: `[H|T]`.

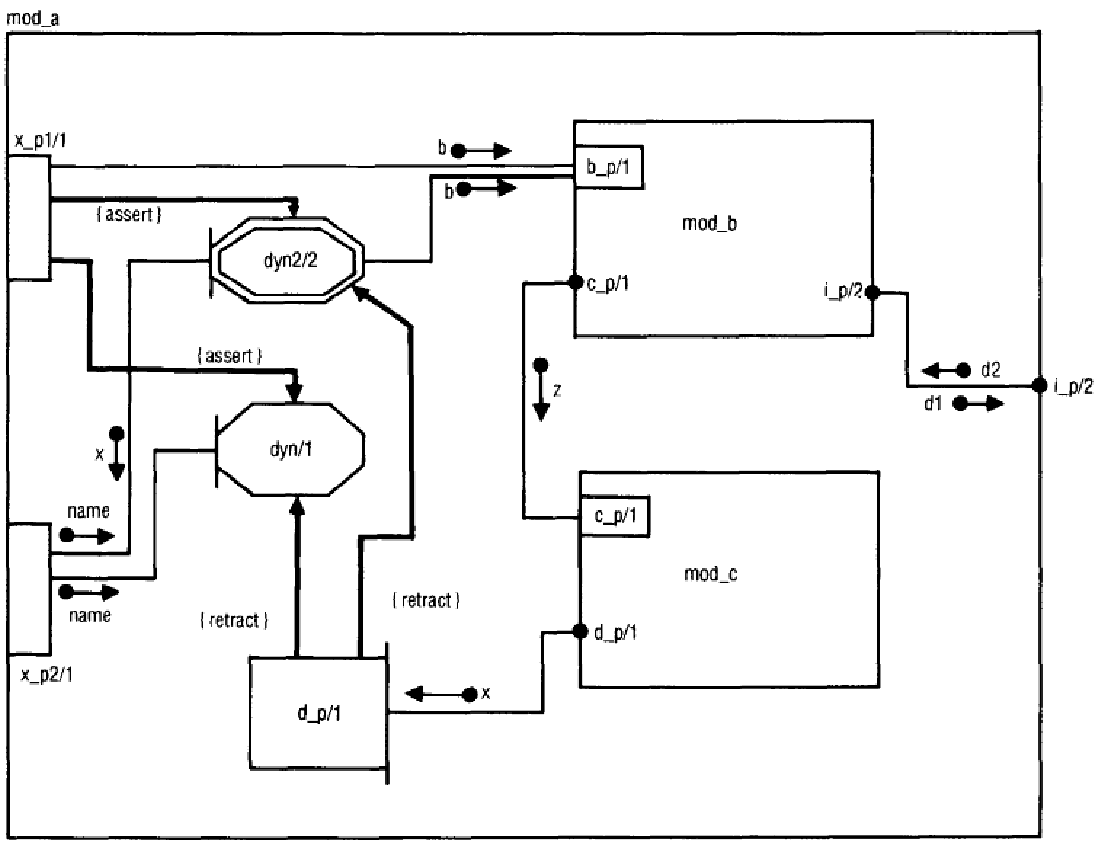
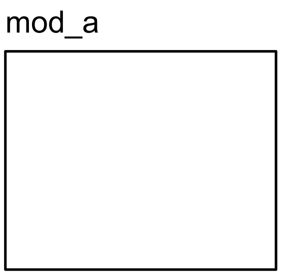


Figure C.2: Example of a internal view diagram of a simple system [24].

The internal view diagram of the Figure C.2 contains the implementation of the module *mod_a* of the system example presented in Figure C.1. Next, the symbols that can be part of these diagrams are shown.

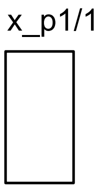
Module



The outer rectangle of an internal view diagram represents the module that is being designed or inspected (from now on, the *main module*).

Within the main module there may be one or several submodules, of which only its external view should be shown.

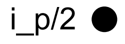
Exported predicate



A rectangle at the edge of the main module symbolizes an exported predicate, visible to the rest of the modules that are at the same level.

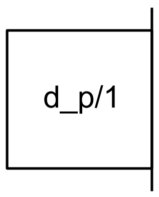
This notation considers that all exported predicates are static, that is, they have an unalterable behavior at runtime.

IMPORTED PREDICATE



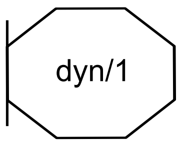
As in external view diagrams, imported predicates are represented by solid circles located at the edge of the main module.

STATIC PREDICATE



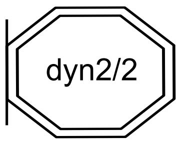
A rectangle with an extended side denotes a static predicate defined within the main module.
In this as in the following classes of predicates, the extended side symbolizes the entry point to which the calls originated by other predicates are connected.

FACT-BASED DYNAMIC PREDICATES



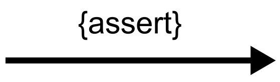
Octagons with a simple edge and an extended side represent «fact-based» dynamic predicates, that is, sets of facts dynamically added to a module through the `assert` predicate.
These predicates can be understood as data repositories that are consulted and modified by other predicates of the same module.

RULE-BASED DYNAMIC PREDICATES



Octagons with a double edge and an extended side represent «rule-based» dynamic predicates, that is, sets of rules added at runtime to a module by means of the `assert` predicate.
Predicates of this type can be understood as variable code portions of a module.

MODIFYING DYNAMIC PREDICATES



The thick arrows indicate what are the only predicates that can modify the clauses of the dynamic predicates.
The text between braces is a notation that serves to clarify if the relationship between both predicates is based on adding (*assert*) and/or eliminating (*retract*) clauses of the dynamic predicate.

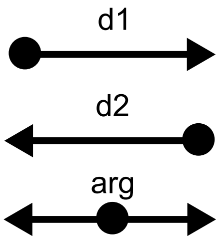
Call

The fine lines allow to represent the calls that are carried out between the different predicates of the main module, as well as between these and the predicates exported and imported from the submodules.

Since these lines are the same at both ends, the following rules must be followed to determine the destination of a call:

- If a line connects any predicate with the entry point of a static or dynamic predicate, this is the destination of the call. Example: *mod_c.d_p/1* $\rightarrow$ *mod_a.d_p/1*.
- If a predicate exported from a submodule is found at the end of a line, it is always the destination. Example: *mod_a.x_p/1* $\rightarrow$ *mod_b.b_p/1*.
- In case the line connects two imported predicates, the destination of the call is the imported predicate from the most external module. Example: *mod_b.i_p/2* $\rightarrow$ *mod_a.i_p/2*

Input, output or input/output parameter



These arrows symbolize the flow of data that occurs in calls between predicates and submodules. The explanation of the three types of arrows can be found in the previous section, page 135.

BIBLIOGRAPHY

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1988.

[2] H. Aït-Kaci. *Warren´s Abstract Machine. A Tutorial Reconstruction*. The MIT Press, 1991.

[3] J. F. Baldwin. Evidential Support Logic Programming. *Fuzzy Sets and Systems*, 24(1):1–26, 1987.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *El Lenguaje Unificado de Modelado*. Addison Wesley, 1999.

[5] R. J. A. Buhr. *System Design with Ada*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[6] R. Caballero, M. Rodríguez-Artalejo, and C. A. Romero-Díaz. Similarity-based Reasoning in Qualified Logic Programming. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and Practice of Declarative Programming, PPDP '08, Valencia, Spain*, pages 185–194. ACM, 2008.

[7] M. Cayrol, H. Farreny, and H. Prade. Fuzzy Pattern Matching. *Kybernetes*, 11(2):103–116, 1982.

[8] F. A. Fontana. Likelog for Flexible Query Answering. *Soft Computing*, 7(2):107–114, 2002.

[9] F. A. Fontana and F. Formato. A Fuzzy Logic Programming Language. In *Proceedings of the APPIA-GULP-PRODE '97*, pages 319–332, 1997.

[10] F. A. Fontana and F. Formato. Likelog: A Logic Programming Language for Flexible Data Retrieval. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, pages 260–267, 1999.

[11] F. Formato. *On Similarity and its application to Logic programming*. PhD thesis, University of Naples Federico II, 1999.

[12] S. Guadarrama, S. Muñoz, and C. Vaucheret. Fuzzy Prolog: A new approach using Soft Constraints Propagation. *Fuzzy Sets and Systems*, 144(1):127–150, 2004.

[13] M. Ishizuka and N. Kanai. Prolog-ELF incorporating Fuzzy Logic. *New Generation Computing*, 3(4):479–486, 1985.

[14] ISO/IEC 13211-1:1995. *Information technology – Programming languages – Prolog – Part 1: General core*. International Organization for Standardization, 1995.

[15] P. Julián and M. Alpuente. *Programación Lógica. Teoría y Práctica*. Pearson Prentice Hall, 2007.

[16] P. Julián and C. Rubio. Introducing Weak Unification into the WAM. Technical report, Dep. of Information Technologies and Systems, University of Castilla-La Mancha, 2006.

[17] P. Julián and C. Rubio. A Declarative Semantics for Bousi-Prolog. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 149–160. ACM, 2009.

[18] P. Julián and C. Rubio. A Programming Environment for Bousi$\sim$Prolog. In *Proceedings of the 2010 International Conference on Artificial Intelligence, ICAI 2010, Las Vegas Nevada, USA, July 12-15*, pages 36–42. CSREA Press, 2010.

[19] P. Julián and C. Rubio. An Efficient Fuzzy Unification Method and its Implementation into the Bousi$\sim$Prolog System. In *IEEE International Conference on Fuzzy Systems (FUZZ)*, pages 1–8. IEEE, 2010.

[20] P. Julián and C. Rubio. Bousi$\sim$Prolog: A Fuzzy Logic Programming Language for Modeling Vague Knowledge and Approximate Reasoning. In *Proceedings of the International Conference on Fuzzy Computation and International Conference on Neural Computation, ICFC-ICNC 2010, Valencia, Spain, October 24-26*, pages 93–98. SciTePress, 2010.

[21] P. Julián, C. Rubio, and J. Gallardo. The Bousi$\sim$Prolog User Manual In A Nutshell (Version 1.04 – Release April 2008). Technical report, Dep. of Information Technologies and Systems, University of Castilla-La Mancha, 2008.

[22] P. Julián, C. Rubio, and J. Gallardo. Bousi∼Prolog: a Prolog Extension Language for Flexible Query Answering. *Electronic Notes in Theoretical Computer Science*, 248:131–147, 2009.

[23] P. Julián, C. Rubio, and J. Gallardo. Inclusión de Conjuntos Borrosos en el Núcleo del Lenguaje Bousi∼Prolog. In *Proceedings of the First Workshop on Computational Logics and Artificial Intelligence, CLAI'2009 (integrated in CAEPIA'2009), Sevilla, Spain, November 9*, pages 81–90. Universidad de Sevilla, 2009.

[24] G. M. Karam. An Icon-Based Design Method for Prolog. *Software, IEEE*, 5(4):51–65, 1988.

[25] M. Kefer and V. S. Subrahmanian. Theory of Generalized Annotated Logic Programming and its Applications. *Journal of Logic Programming*, 12:335–367, 1992.

[26] V. Loia, S. Senatore, and M. I. Sessa. Similarity-based SLD Resolution and its implementation in an Extended Prolog System. In *The 10th IEEE International Conference on Fuzzy Systems*, volume 2, pages 650–653. IEEE, 2001.

[27] V. Loia, S. Senatore, and M. I. Sessa. Similarity-based SLD Resolution and its role for Web Knowledge Discovery. *Fuzzy Sets and Systems*, 144(1):151–171, 2004.

[28] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[29] T. P. Martin, J. F. Baldwin, and B. W. Pilsworth. The Implementation of FProlog – A Fuzzy Prolog Interpreter. *Fuzzy Sets and Systems*, 23(1):119–129, 1987.

[30] M. Mukaidono, Z. L. Shen, and L. Ding. Fundamentals of Fuzzy Prolog. *International Journal of Approximate Reasoning*, 3(2):179–193, 1989.

[31] H. T. Nguyen and E. A. Walker. *A First Course in Fuzzy Logic*. CRC Press, 2000.

[32] R. A. O'Keefe. *The Craft of Prolog*. The MIT Press, 1994.

[33] R. A. Orchard. *FuzzyCLIPS Version 6.10d. User's Guide*, 2004.

[34] F. Pascual, P. Julián, M. Ferreira, and J. Gallardo. Una Aproximación Declarativa a la Clasificación de Documentos. In *XV Congreso Español de Tecnología y Lógica Fuzzy,*

ESTYLF 2010, Huelva, Spain, February 3-5, pages 543–548. Universidad de Huelva, 2010.

[35] W. Pedrycz and F. Gomide. *An Introduction to Fuzzy Sets: Analysis and Design*. The MIT Press, 1998.

[36] R. Pérez. *Procesado y Optimización de Espectros Raman mediante Técnicas de Lógica Difusa: Aplicación a la identificación de Materiales Pictóricos*. PhD thesis, Universitat Politècnica de Catalunya, 2005.

[37] L. G. Rios-Filho and S. A. Sandri. Contextual Fuzzy Unification. In *5th IFSA*, volume 95, pages 81–84, 1995.

[38] C. Rubio. *Diseño e Implementación de un Lenguaje de Programación Lógica Borrosa con Unificación Débil*. PhD thesis, Universidad de Castilla-La Mancha, 2011.

[39] M. I. Sessa. Approximate Reasoning by Similarity-based SLD Resolution. *Theoretical Computer Science*, 275(1-2):389–426, 2002.

[40] L. Sterling and E. Y. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1994.

[41] H. E. Virtanen. Fuzzy Unification. In *Proceedings of the International Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems (IPMU 1994)*, pages 1147–1152, 1991.

[42] H. E. Virtanen. Linguistic Logic Programming. *Logic Programming and Soft Computing*, pages 91–128, 1998.

[43] P. Vojtáš. Fuzzy Logic Programming. *Fuzzy Sets and Systems*, 124(3):361–370, 2001.

[44] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965.

[45] L. A. Zadeh. The Concept of a Linguistic Variable and its Application to Approximate Reasoning, Parts I, II and III. *Information Sciences*, 8 and 9, 1975.

[46] L. A. Zadeh. Fuzzy Logic. *Computer*, 21(4):93, 1988.