UNICORN: A Programming Environment for Bousi∼Prolog.

Pascual Julián-Iranzo[1] Clemente Rubio-Manzano[2]

Dep. of Information Technologies and Systems,
University of Castilla-La Mancha,
Spain.

Abstract

Bousi∼Prolog is a fuzzy logic programming language which is an extension of the standard Prolog language. The Bousi∼Prolog operational semantics is an adaptation of the SLD resolution principle, where classical unification has been replaced by a fuzzy unification algorithm based on proximity relations. UNICORN is a programming environment for the Bousi∼Prolog language. This environment will provide resources to **edit, save, open, compile and run** Bousi∼Prolog programs comfortably, in addition to various options for supporting programmers: commands, directives and visualization windows. In this paper we summarize the main features of the UNICORN environment and we provide some insides about its implementation.

Keywords: Fuzzy Logic Programming, Fuzzy Prolog, Unification by Similarity, Weak SLD Resolution, Warren Abstract Machine.

1 Introduction.

Bousi∼Prolog (BPL, for short) [4] is an extension of the standard Prolog language. Its operational semantics is an adaptation of the SLD resolution principle where classical unification has been replaced by a fuzzy unification algorithm based on proximity relations defined on a syntactic domain. Proximity relations are fuzzy relations that fulfill the reflexive and symmetric properties [3]. Hence, the operational mechanism is a generalization of the similarity-based SLD resolution principle [9]. Informally, this weak unification algorithm states that two terms $f(t_1, \ldots, t_n)$ and $g(s_1, \ldots, s_n)$ weakly unify if the root symbols f and g are approximate and each of their arguments t_i and s_i weakly unify. Therefore, the weak unification algorithm does not produce a failure when there is a clash of two syntactical distinct symbols,

⋆ This work has been partially supported by FEDER and the Spanish Science and Innovation Ministry under grant TIN 2007-65749 and by the Castilla-La Mancha Administration under grant PII1I09-0117-4481.

[1] Email: Pascual.Julian@uclm.es

[2] Email: Clemente.Rubio@alu.uclm.es

[3] When, in addition, a fuzzy binary relation fulfills the transitive property, we are in presence of a similarity relation.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

whenever they are approximate, but a success with a certain approximation degree. Hence, Bousi∼Prolog computes substitutions as well as approximation degrees.

The aim of Bousi∼Prolog is the management of uncertainty using declarative techniques. As it was shown in [4], there exist several practical applications for a language with the aforementioned characteristics: flexible query answering; advanced pattern matching; information retrieval where textual information is selected or analyzed using an ontology; text cataloging and analysis; etc.

Bousi∼Prolog is publicly available and can be found at the URL address `http://www.inf-cr.uclm.es/www/pjulian/bousi.html`. Currently it is delivered in two implementation formats: a high level and a low level implementation. The high level implementation [4] is written in Prolog through a meta-interpreter. Hence it may present efficiency problems. In order to solve them, we have investigated how to incorporate the weak unification algorithm into the Warren Abstract Machine (WAM) [11], leading to a low level implementation of the Bousi∼Prolog language, which consists of an architecture compounded by a compiler and an enlargement of the WAM able to execute BPL programs efficiently. The extension of the WAM able of handling fuzzy relations is called Similarity-based WAM (SWAM) by historical reasons (although the current implementation allows the treatment of proximity relations as well as similarity relations). The SWAM was implemented mainly using the standard techniques of [1] and it is described in [5].

To allow the development of BPL programs, a programming environment for this language, called Unicorn, has been created. This environment will provide resources to **edit**, **save**, **open**, **compile** and **run** BPL programs in a comfortable, easy, usable way, in addition to various options for supporting programmers: commands, directives and visualization windows, that help the programmer to obtain a better understanding of the BPL programs behavior. In this paper we summarize the main features of the Unicorn environment and we provide some insides about its implementation. As imperative and object oriented languages have really good integrated development environments (IDEs) with graphical user interfaces that allow people to interact through direct manipulation of graphical elements, we take inspiration on these interfaces to accomplish our objective.

Before ending this introductory section let us comment some pieces of related work: The first one is represented by the theoretical work [3], whose main practical realization is the fuzzy logic language LIKELOG [2] (an interpreter implemented in Prolog using rather direct techniques and cumbersome concepts). The second line of research is the closest to ours and it is based on the theoretical work [9]. In [6], a similarity-based logic programming language, named SiLog, was presented. SiLog is an interpreter written in Java. Both approaches are based on an extension of the SLD resolution principle with the ability of dealing with similarity relations.

Neither LIKELOG nor SiLog are publicly available and therefore a practical comparison is impossible. However, we can enumerate three important features of the Bousi∼Prolog language that distinguish it from these other proposals:

(i) Bousi∼Prolog uses proximity relations (as well as similarity relations), therefore its operational semantics is more general and flexible than the ones used by LIKELOG and SiLog which are exclusively based on similarity relations.

(ii) In order to obtain a proximity or a similarity relation, Bousi∼Prolog gives automatic support to the user for the construction of a reflexive, symmetric and/or transitive closure, starting from an arbitrary fuzzy binary relation.

(iii) Bousi∼Prolog is a true Prolog extension and not a simple interpreter able to execute a weak SLD resolution procedure.

2 The Bousi∼Prolog Programming Language.

In this section we briefly summarize the features of Bousi∼Prolog as it has been implemented in the present version supported by the SWAM. We concentrate on the syntactical and operational aspects.

As it was just commented, the BPL programming language is an extension of the standard Prolog language with a proximity/similarity relation defined on a syntactic domain. Therefore, the syntax is mainly the Prolog syntax but enriched with a built-in symbol "∼" used for describing proximity/similarity relations (actually, fuzzy binary relations which are automatically converted into proximity/similarity relations) by means of *similarity equations* of the form:

```
<alphabet symbol> ~ <alphabet symbol> = <similarity degree>
```

Although, a similarity equation represents an arbitrary fuzzy binary relation, its intuitive reading is that two constants, n-ary function symbols or n-ary predicate symbols are approximate or similar with a certain degree. That is, a similarity equation $a \sim b = \alpha$ can be understood in both directions: a is approximate/similar to b and b is approximate/similar to a with degree α. Therefore, a Bousi∼Prolog program is a sequence of Prolog facts and rules followed by a sequence of similarity equations.

Example 2.1 Suppose a fragment of a database that stores a semantic network with information about people's names and hair color, as well as the approximate relation between black, brown and blond hair[4].

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

% FACTS                                          % SIMILARITY EQUATIONS
is_a(john, person).     hair_color(john,black).     black~brown=0.6.
is_a(peter, person).    hair_color(peter,brown).    black~blond=0.3.
is_a(mary, person).     hair_color(mary,blond).     blond~brown=0.6.
```

In a standard Prolog system, if we ask about whether `peter`'s hair is `blond`, "`?- hair_color(peter, blond)`", the system fails. However BPL allows us to obtain the answer "`Yes with 0.6`".

To obtain this answer, the BPL system operates as follows:

i) First it generates the reflexive, symmetric closure of the fuzzy relation $\{\mathcal{R}(black, brown) = 0.6,\ \mathcal{R}(black, blond) = 0.3,\ \mathcal{R}(blond, brown) = 0.6\}$, constructing a proximity relation. This is done at compilation time.

ii) Then, at execution time, it tries to unify the goal `hair_color(peter, blond)` and (eventually) the fact `hair_color(peter,brown)`. Because there exists the entry $\mathcal{R}(brown, blond) = 0.6$ in the constructed proximity relation (that is, `brown`

[4] For the sake of simplicity we only consider programs without variables in the examples. Of course, BPL permits programs and goals containing variables.

is approximate to `blond`), the unification process succeeds with approximation degree 0.6 and a (weak) resolution step is done, leading to the empty clause.

The above example serves to illustrate both the syntax and the operational semantics of the language. Also, it is important to note that, in this example, to inhibit the construction of the transitive closure (and therefore the construction of a similarity relation) has been crucial to model the information properly and to obtain a convenient result. This effect is obtained by the BPL directive `transitivity` which disables or enables the construction of the transitive closure of a fuzzy relation during the compilation process [5]. If a similarity relation would be generated, the system would constructed the entries $\mathcal{R}(brown, blond) = 0.6$ and $\mathcal{R}(blond, brown) = 0.6$, overlapping the initial approximation degree provided by the user and leading to a wrong information modeling [6]. Therefore, as it was commented in [10] and this example confirms, similarity relations sometimes represent fuzzy information incorrectly. Hence, to allow the use of proximity relations, not only increases the expressive power of the language, but it is critical in order to solve certain problems.

On the other hand, Bousi~Prolog implements a weak unification operator, also denoted by "~", which is the fuzzy counterpart of the syntactical unification operator "=" of standard Prolog. It can be used, in the source language, to construct expressions like "`Term1 ~ Term2 =:= Degree`" which is interpreted as follows: The expression is true if `Term1` and `Term2` are unifiable by similarity with approximation degree `AD` equal to `Degree`. In general, we can construct expressions

```
Term1 ~ Term2 <op> Degree
```

where "`<op>`" is a arithmetic comparison operator (that is, an operator in the set `{=:=, =\=, >, <, >=, =<}`). Observe that the expression "`Term1 ~ Term2`" is syntactic sugar of "`Term1 ~ Term2 > 0`". These expressions may be introduced in a query as well as in the body of a clause.

Finally the BPL system implementation covers the main features of standard Prolog: arithmetic, lists, cut operator, input/output (read and write), negation (predicate not), also it has some built-in predicates like assert and retract.

3 Requirements and Installation Procedure.

If you want to install the BPL system with the UNICORN environment on your computer, you need a computer running Windows 2000/XP, Linux or Mac/OS operating system and the Java Virtual Machine version 1.5 (JVM 1.5) installed on it.

Therefore, you must follow these simple steps:

(i) Check if the JVM 1.5 is installed. If you need to install JVM 1.5, go to the URL address: `http://java.sun.com/javase/downloads/index_jdk5.jsp`.

(ii) Download the file "`UNICORN_1.0_beta.jar`" into the BPL home directory.

[5] Note that in the current implementation of the BPL system, `transitivity(no)` is the current default option.

[6] Because a person with brown hair should be closer to a person with blond hair than a person with black hair is to a person with blond hair.

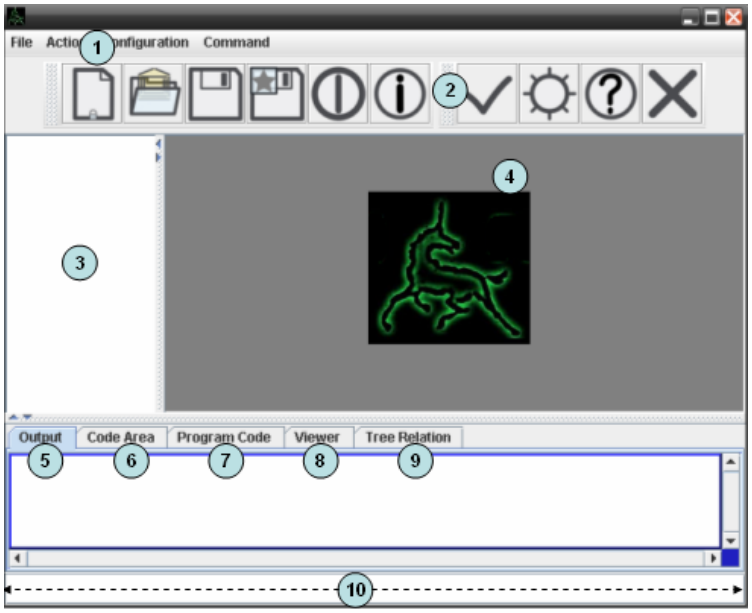

Fig. 1. The UNICORN programming environment: main window.

(iii) Execute it, according with the usual procedure used in that operating system.

Then the UNICORN environment is available (see Figure 1).

4 The UNICORN Environment.

In this section we give a brief tutorial on how to use the UNICORN environment, which is the interface between the user and the BPL system.

The UNICORN environment is divided in two different zones: the command options zone (with the Menu Bar (1) and the Icon Bar (2) at the top of the screen) and the windows zone. The windows zone involves five kinds of windows:

- The **query window** (10) placed at the bottom of the screen, serves to introduce queries, commands or data to the system.
- The **output window** (5) shows the answers to a query and other system information. You can delete the information in the output window, by means of the command `clear` written into the query window.
- The **code area window** (6) shows the SWAM machine code obtained after the compilation of a source program. It is the object program executed by the abstract machine. For instance, Figure 2 shows the SWAM machine code obtained for the BPL program of the Example 2.1. You can delete the information in the code area window, by typing the command `reset` into the query window.
- The **code execution window** (7) shows, sequentially, the SWAM machine code executed by the abstract machine. It can be seen as a tracer tool.
- The **visualization window** (3,8,9) shows a graph representation (8) of the proximity relation defined in the program. Also, it is shown a pictorial representation (3) of the *Proximity Matrix*, that is, an adjacency matrix representation of the reflexive, symmetric, (and possibly) transitive closure of the original fuzzy binary

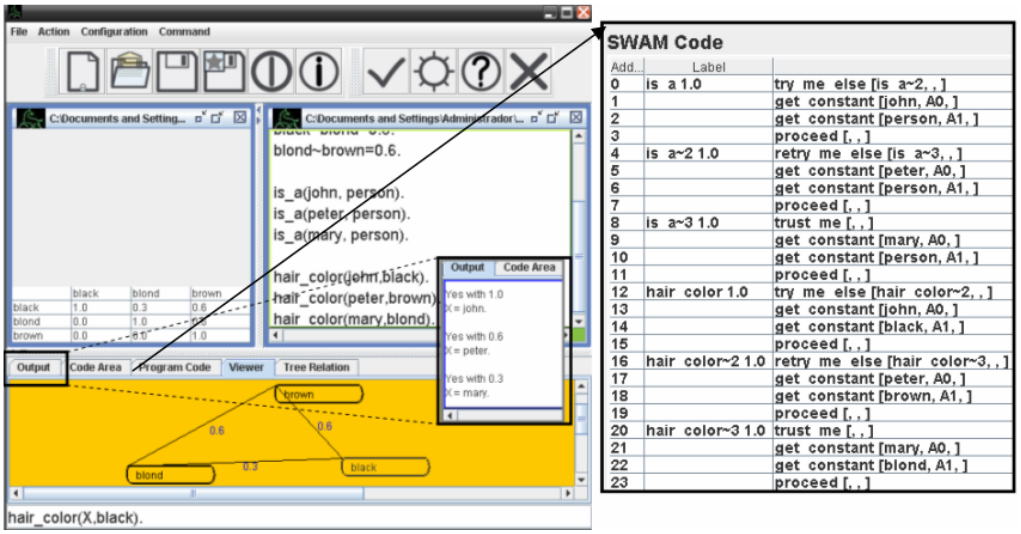

Fig. 2. Visualization windows, Output window and Code Area Window for the BPL program of Example 2.1.

ACTIONS	OPTIONS		
	File Menu	Icon Bar	Query Window
Creating new program	New	White paper Icon	"crtl-N"
Save a program (Home directory)	Save	Diskette Icon	"ctrl-G"
Save a program (Other directory)	Save As	Star diskette Icon	"ctrl-M"
Editing a program	Open	Opening Archive Icon	"ctrl-A"
Compilation of a program	Compile	OK Icon	"ctrl-C"
Execution of a program	Execute	Sun Icon	"ctrl-E"
More responses	;	Interrogation Icon	"ctrl-S"
Stop execution	Stop	Cross Icon	"ctrl-P"

Fig. 3. Options for edit, save, compile and execute BPL programs.

relation (which is computed starting from the set of proximity equations provided by the program). Figure 2 shows each type of visualization window with more detail.

As we shall comment, it is also possible to open several **edit windows** (4) to create or modify programs.

The UNICORN environment provides resources to **edit, save, open, compile and run** BPL programs comfortably. Figure 3 summarizes the options that you can use to perform these actions. Moreover, it has several commands useful to reset the information shown by the output window and the code area window or to produce some actions or activate some internal flags.

(i) The command "`compile`" which compiles the selected program.

(ii) The command "`clear`" deletes the information shown in the output window.

(iii) The command "`reset`" that resets the memory layout and deletes the information shown in the code area window.

(iv) The command "`statistics`" returns the time used in the last execution in milliseconds.

(v) The command "`lambdaCut`" imposes a limit to the expansion of the search

space in a computation [7]. By typing "`lambdaCut(N)`" into the query window, you set the value of the internal lambdacut flag to `N` (being `N` a number between 0 and 1). When the lambdacut flag is set to `N`, the weak unification process fails if the computed approximation degree goes below the stored lambdacut value `N`. Therefore, the computation also fails and all possible branches starting from that choice point are discarded. By default the lambdacut flag is set to zero (that is, no limitation is imposed). The following example illustrates the use of the command lambda cut.

Example 4.1 Revisiting the Example 2.1, suppose that we want to fix the lambdacut flag to `0.5`. Then you must write "`lambdaCut(0.5).`" into the query window, being "`Lambdacut fixed to 0.5.`" the resulting message. Therefore, if we ask again by the hair color of a person (writting the goal "`hair_color(X,black).`" into the query window) the system answers "`X=john with 1.0`" and "`X=peter with 0.6`", but the answer "`X=mary with 0.3`", whose approximation degree is lower than `0.5`, is not obtained.

5 Design and Implementation of the UNICORN Environment.

The architecture of the BPL low level implementation is a multi-layer architecture with three layers: the Unicorn environment, the compiler and the similarity abstract machine (SWAM). It consists of over 6500 code lines, divided into 27 classes. It has been implemented in Java, since this is an object oriented language that possesses facilities to deploy the BPL system on the web. The BPL architecture is depicted in Figure 4.

5.1 Similarity-based WAM.

The Similarity-based WAM (SWAM) modifies the two main parts of a standard WAM: the memory layout and the instruction set [5]. The SWAM is executed as a thread inherited from the class `Thread`. The idea is to allow a programmer to use interactive predicates or I/O predicates like, for instance, the predicate `read`, for which it is necessary that the SWAM keeps waiting while the user writes the input data.

Extension of the memory layout. The main changes are related to the incorporation of data structures that allow us to manage proximity or similarity relations. We add the so called *Proximity Matrix Area*, which stores an adjacency matrix representation of a proximity or similarity relation. Two new specific registers: the *Aproximation Degree register* (`AD`), which stores the current computed approximation degree of a derivation; and the *Lambda-Cut register* (`LC`) which stores the lower bound for the approximation degree in a derivation. Also, we need to modify the standard choice point frame structure by adding a new field, `D`, to save the value stored in the `AD` register, prior to the creation of a choice point.

[7] Observe that this limit can also be modified using the BPL directive ":-lambdaCut(N)."

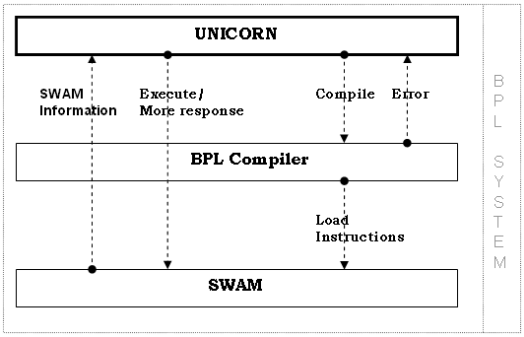

Fig. 4. The BPL system and the Unicorn environment.

Extension of the instruction set. As the choice point frame structure has been modified, the machine instructions that work in combination with it need also to be modified. That is, we have to modify the instructions: `try_me_else`, `retry_me_else`, `trust_me` and `backtrack`. The first three machine instructions essentially update the value of the `AD` register and `backtrack` helps to recover the old approximation degree value after a failure. On the other hand, also it is necessary to modify some machine instructions involved in the process of argument unification, such as: `get_structure`, `get_constant` and `unify`.

5.2 BPL compiler.

The structure of the BPL compiler has four main parts, being the SWAM machine the basis for the compiler implementation: i) given a source program, the **Analyzer** verifies that the programs are syntactically correct and obtains the syntactic tree which is the basis for later code generation; ii) the **Similarity Generator** calculates a proximity relation or a similarity relation (if the transitivity option is enabled) starting from the similarity equations provided by the programmer; the proximity/similarity relation is stored into the Proximity Matrix memory area and its information is used at compilation time, by the Adapter, and at execution time, when it is necessary during the unification process. iii) the **Adapter** takes the syntactic tree and the proximity or similarity relation and constructs an intermediate representation which is used by the Code Generator to obtain the object code; iv) finally, the **Code Generator** produces the machine code associated to the source program; once the machine code is generated, it is stored in the Code Area, an addressable array of memory words.

5.3 The Unicorn environment.

The Unicorn environment is the top layer of the BPL system. It has been mainly implemented with the *swing* and the *awt* Java 1.5 packages [8], using *Netbeans* 5.5 as development environment [7]. It possesses a clean object oriented design. It consists of over 1500 code lines, divided into 6 classes. Figure 5 shows a simplified view for the UML class diagram of the Unicorn environment.

In the following we give a summary with the features of the main classes involved in the implementation of the Unicorn environment, according with the UML description of Figure 5:

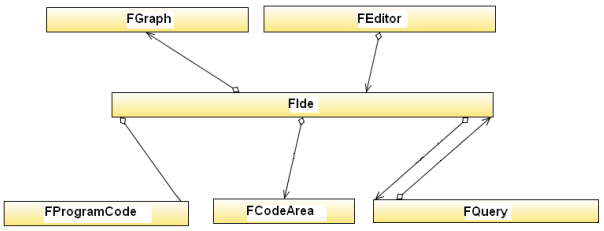

Fig. 5. A simplified class diagram for the Unicorn environment.

- FIde is a frame that extends the class `JFrame` of the *Swing* package. It is the class implementing the Unicorn graphical interface. It provides the main window and the remainder windows are associated to this class.
- FEditor is the class in charged of implementing the Unicorneditor. To fulfill this task we mainly use the classes `JInternalFrame` and `JTextPanel` of the *Swing* package for the graphical components, and `FileReader` or `FileWritter` of the *IO* package, for operating with the file system. This class implements the whole functionality for managing archives through the methods: `openFile()`, `writeFile()`, `save()`, `saveAs()`.
- FQuery is the class implementing the input command line of the system. That is, it enables the input for queries, commands and data. We mainly use the class `JTextField` of the *Swing* package in its implementation. It contains important methods to synchronize the input data with the SWAM: `wait()` which put the SWAM into a waiting state until its resource becomes available; `continue()` which tells the SWAM that the data have been introduced or the resource is available and therefore the execution can continue.
- FCodeArea is the class that shows the contain of the SWAM code area. It is instanced when the compilation process ends successfully. To implement this class we mainly use the class `JTable` of the *Swing* package. It contains the method `loadInstructions()` which loads in a `JTable` object (i.e., a graphical table) the machine instructions stored in the SWAM code area.
- FProgramCode shows the executed machine instructions step by step. It can be seen as a tracer tool. In case of error it points the instruction that produces the error. This class has been implemented using the class `JTextArea` of the *Swing* package. It is instanced when the object program is executed by the SWAM. It contains the method `writeProgramCode()` that writes the machine instructions, as they are executed, in a `JTextArea` object, in order to their visualization.
- FGraph is a class that shows a graphical representation of the proximity relation defined in the program. It is instanced in compilation time when the reflexive, symmetric, (or possibly) transitive closure (if the internal transitivity flag is set to `yes`). In its implementation we employ the class `JPanel` of the *Swing* package (we use the standard methods `update(Graphics)` and `paintComponent(Graphics)`). It contains the methods `moveVertice()` and `pressVertice()` which allow the user select and move the graph vertices to visualize it correctly.

6 Conclusions and Future Work.

In this paper we presented the Unicorn environment. It is a programming environment to facilitate the development of BPL programs. It was mainly implemented using the *swing* and the *awt* Java 1.5 packages with a clean object oriented design. This environment integrates resources to **edit**, **save**, **open**, **compile** and **run** BPL programs in a comfortable, easy, usable way. In addition it provides various tools for supporting programmers and help them to obtain a better understanding of programs behavior:

- A code area window that shows the SWAM machine code obtained after the compilation of a source program.
- A tracer tool that shows, sequentially, the SWAM machine code as it is executed by the abstract machine.
- A tool that shows a graph representation of the proximity relation defined by the program.

An integrated development environment (IDE) additionally to a source code specialized editor (to facilitate the program writing) and a compiler (or interpreter), normally consists of a debugger and build automation tools. As a matter of future work we want to extend the Unicorn environment with these kinds of tool and features. The aim is to convert it into a true IDE.

References

[1] Aït-Kaci, H: *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, MA (1991).

[2] Fontana, F., Formato, F.: Likelog: A logic programming language for flexible data retrieval. In: Proc. of the ACM SAC, pp. 260–267 (1999).

[3] Fontana, F., Formato, F.: A similarity-based resolution rule. Int. J. Intell. Syst. 17(9), 853–872 (2002).

[4] Julián, P., Rubio, C., Gallardo. J.: Bousi∼prolog: a prolog extension language for flexible query answering. In: ENTCS, p. 16. Elsevier, Amsterdam (in press, 2009).

[5] Julián, P., Rubio, C.: A similarity-based WAM for Bousi∼Prolog. In: LNCS, vol 5517, pp. 245–252. Springer, Heidelberg (2009).

[6] Loia, V., Senatore, S., Sessa, M.I.: Similarity-based SLD resolution and its implementation in an extended prolog system. In FUZZ-IEEE, pp. 650–653 (2001).

[7] Sun MycroSystems. NetBeans IDE. Available at: `http://www.netbeans.org/` (2000).

[8] Sun MycroSystems. API Java 2. Available at: `http://java.sun.com/j2se/1.5.0/docs/api/` (2004).

[9] Sessa, M.I.: Approximate reasoning by similarity-based SLD resolution. *Theoretical Computer Science*, 275(1-2), 389–426 (2002).

[10] Shenoi, S., Melton, A.: Proximity relations in the fuzzy relational database model. *Fuzzy Sets and Systems*, 100, 51–62 (1999).

[11] Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical note 309, SRI International, Menlo Park, CA., October (1983).