

FUZZY LOGIC DESIGN TOOLS

X fuzzy 3.0

©IMSE-CNM 1997-2003

Xfuzzy is property of its authors and IMSE-CNM

Xfuzzy is free software; you can redistribute it and/or modify it under the terms of the [GNU General Public License](#) as published by the [Free Software Foundation](#).

Xfuzzy is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Table of Contents

- Release notes for version 3.0 4
- An Overview of Xfuzzy 3.0 5
- Installation of Xfuzzy 3.0 6
- XFL3: The Xfuzzy 3.0 specification language 7
 - Operator sets
 - Types of linguistic variables
 - Rule bases
 - System global behavior
 - Function packages
 - Binary function definition
 - Unary function definition
 - Membership function definition
 - Defuzzification method definition
 - The standard package xfl
- The Xfuzzy 3.0 development environment 25
 - Description stage
 - System edition (xfedit)
 - Package edition (xfpkg)
 - Verification stage
 - 2-dimensional representation (xf2dplot)
 - 3-dimensional representation (xf3dplot)
 - Inference monitor (xfmt)
 - System simulation (xfsim)
 - Tuning stage
 - Supervised learning (xfsl)
 - Synthesis stage
 - C code generator (xfc)
 - C++ code generator (xfcpp)
 - Java code generator (xfj)

Release Notes for version 3.0

Changes in version 3.0.0 with respect to 2.X

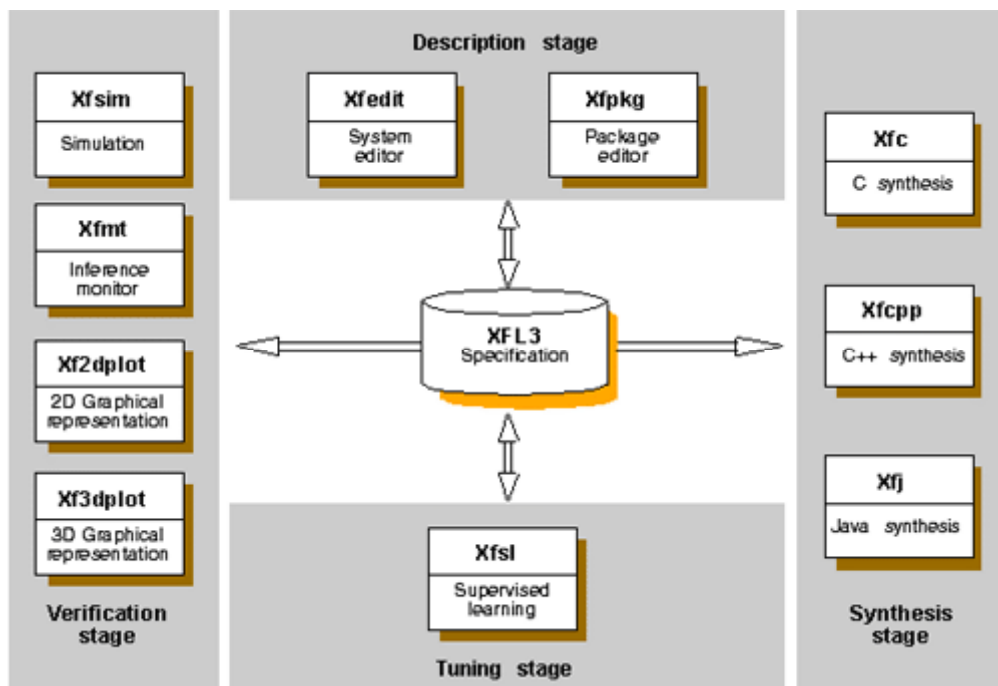
1. The environment has been completely reprogrammed using Java.
2. A new specification language, called XFL3, has been defined. Some of the improvements with respect to XFL are the following:
 1. A new kind of object, called operator set, has been incorporated to assign different functions to the fuzzy operators.
 2. Linguistic hedges, which describe more complex relationships among the linguistic variables, have also been included.
 3. User can now extend not only the functions assigned to fuzzy connectives and defuzzification methods, but also to membership functions and linguistic hedges.
3. The edition tool can now define hierarchical rule bases.
4. The 2-D and 3-D representation tools do not require gnuplot.
5. A new monitor tool has been added to study the system behavior.
6. The learning tool includes many new learning algorithms.

Known bugs in version 3.0

1. (xfedit) Membership functions edition sometimes provokes the error "Label already exists".
2. (xfedit) Rulebases edition produces an error upon applying the modifications twice.
3. (xfedit, xfmt) The hierarchical structure of the system is not drawn correctly when an internal variable is used both as input to the rulebase and as output variable
4. (xfsim) The end conditions upon the system input variables are not correctly verified.
5. (tools) The command-mode execution of the different tools does not admit absolute path to identify files.
6. (XFL3) The "definedfor" clause is not verified by the defuzzification methods.
7. (xfcpp) Some compilers do not admit that the methods of the class Operatorset be called "and", "or" or "not".
8. (xfsl) The clustering process may generate new membership functions whose parameters do not comply with the restrictions due to rounding errors.
9. (tools) Sometimes some windows of the tools are not drawn correctly and it is necessary to modify the size of these windows to force a correct representation.

An Overview of Xfuzzy 3.0

Xfuzzy 3.0 is a development environment for fuzzy-inference-based systems. It is composed of several tools that cover the different stages of the fuzzy system design process, from their initial description to the final implementation. Its main features are the capability for developing complex systems and the flexibility of allowing the user to extend the set of available functions. The environment has been completely programmed in Java, so it can be executed on any platform with JRE (Java Runtime Environment) installed. The next figure shows the design flow of Xfuzzy 3.0.



The [description stage](#) includes graphical tools for the fuzzy system definition. The [verification stage](#) is composed of tools for simulation, monitoring and representing graphically the system behavior. The [tuning stage](#) consists in applying learning algorithms. Finally, the [synthesis stage](#) is divided into tools generating high-level languages descriptions for software or hardware implementations.

The nexus between all these tools is the use of a common specification language, [XFL3](#), which extends the capabilities of XFL, the language defined in version 2.0. XFL3 is a flexible and powerful language, which allows to express very complex relations between the fuzzy variables, by means of hierarchical rule bases and user-defined fuzzy connectives, linguistic hedges, membership functions and defuzzification methods.

Every tool can be executed as an independent program. The environment integrates all of them under a [graphical user interface](#) which eases the design process.

Installation of Xfuzzy 3.0

System requirements:

Xfuzzy 3.0 can be executed on platforms containing the Java Runtime Environment. For defining new function packages, a Java Compiler is also needed. The Java Software Development Kit, including JRE, compiler and many other tools can be found at <http://java.sun.com/j2se/>

Installation guide:

- Download the [install.class](#) file.
- Execute this class file with the command "java install". This will open the following window:



- Choose a directory to install Xfuzzy. If this directory does not exist, it will be created in the installation process.
- Click on the Install button. This will uncompress the Xfuzzy distribution on the selected base directory.
- The Xfuzzy executables are located in the "/bin" directory. Add this directory to the PATH environment variable.
- The executable files are script programs. Do not change the location of the Xfuzzy distribution, otherwise these script files will not work.

XFL3: The Xfuzzy 3.0 specification language

- [XFL3: The Xfuzzy 3.0 specification language](#)
 - [Operator sets](#)
 - [Types of linguistic variables](#)
 - [Rule bases](#)
 - [System global behavior](#)
 - [Function packages](#)
 - [Binary function definition](#)
 - [Unary function definition](#)
 - [Membership function definition](#)
 - [Defuzzification method definition](#)
 - [The standard package xfl](#)

Formal languages are usually defined for the specification of fuzzy systems because of its several advantages. However, two objectives may conflict. A generic and high expressive language, able to apply all the fuzzy logic-based formalisms, is desired, but, at the same time, the (possible) constraints of the final system implementation have to be considered. In this sense, some languages focus on expressiveness, while others are focused on software or hardware implementations.

One of our main objectives when we began to develop a fuzzy system design environment was to obtain an open environment that was not constrained by the implementation details, but offered the user a wide set of tools allowing different implementations from a general system description. This led us to the definition of the formal language XFL. The main features of XFL were the separation of the system structure definition from the definition of the functions assigned to the fuzzy operators, and the capabilities for defining complex systems. XFL is the base for several hardware- and software-oriented development tools that constitute the Xfuzzy 2.0 design environment.

As a starting point for the 3.0 version of Xfuzzy, a new language, XFL3, which extends the advantages of XFL, has been defined. XFL3 allows the user to define new membership functions and parametric operators, and admits the use of linguistic hedges that permit to describe more complex relationships among variables. In order to incorporate these improvements, some modifications have been made in the XFL syntax. In addition, the new language XFL3, together with the tools based on it, employ Java as programming language. This means the use of an advantageous object-oriented methodology and the flexibility of executing the new version of Xfuzzy in any platform with JRE (Java Runtime Environment) installed.

XFL3 divides the description of a fuzzy system into two parts: the logical definition of the system structure, which is included in files with extension ".xfl", and the mathematical definition of the fuzzy functions, which are included in files with extension ".pkg" (packages).

The language allows the definition of complex systems. It does not limit the number of linguistic variables, membership functions, fuzzy rules, etc. Systems can be

defined by hierarchical rule bases, and fuzzy rules can express complex relationships among the linguistic variables by using connectives AND and OR, and linguistic hedges like greater than, smaller than, not equal to, etc. XFL3 allows the user to define its own fuzzy functions by means of [packages](#). These new functions can be used as membership functions, fuzzy connectives, linguistic hedges and defuzzification methods. The standard package [xfl](#) contains the most usual functions.

The description of a fuzzy system structure, included in ".xfl" files, employs a formal syntax based on 8 reserved words: *import*, *operatorset*, *type*, *extends*, *rulebase*, *using*, *if* and *system*. An XFL3 specification consists of several objects defining operator sets, variable types, rule bases, and the description of the system global behavior. An [operator set](#) describes the selection of the functions assigned to the different fuzzy operators. A [variable type](#) contains the definition of the universe of discourse, linguistic labels and membership functions related to a linguistic variable. A [rule base](#) defines the logical relationship among the linguistic variables, and, finally, the [system global behavior](#) includes the description of the rule base hierarchy.

Operator sets

An operator set in XFL3 is an object containing the mathematical functions that are assigned to each fuzzy operator. Fuzzy operators can be binary (like the T-norms and S-norms employed to represent linguistic variable connections, implication, or rule aggregations), unary (like the C-norms or the operators related with linguistic hedges), or can be associated with defuzzification methods.

XFL3 defines the operator sets with the following format:

```
operatorset identifier {
    operator assigned_function(parameter_list);
    operator assigned_function(parameter_list);
    ..... }
```

It is not required to specify all the operators. When one of them is not defined, its default function is assumed. The following table shows the operators (and their default functions) currently used in XFL3.

Operator	Type	Default function
and	binary	min(a,b)
or	binary	max(a,b)
implication, imp	binary	min(a,b)
also	binary	max(a,b)
not	unary	(1-a)
very, strongly	unary	a ²
moreorless	unary	(a) ^(1/2)
slightly	unary	4*a*(1-a)
defuzzification, defuz	defuzzification	center of area

The assigned functions are defined in external files which we name as packages. The format to identify a function is "*package.function*". The package name, "xfl" in the following example, can be removed if the package has been imported previously (using the command "*import package;*").

```
operatorset systemop {
  and xfl.min();
  or xfl.max();
  imp xfl.min();
  strongly xfl.pow(3);
  moreorless xfl.pow(0.4);
}
```

Types of linguistic variables

An XFL3 type is an object that describes a type of linguistic variable. This means to define its universe of discourse, to name the linguistic labels covering that universe, and to specify the membership function associated to each label. The definition format of a type is as follows:

```
type identifier [min, max; card] {
  label membership_function(parameter_list);
  label membership_function(parameter_list);
  ..... }
```

where *min* and *max* are the limits of the universe of discourse and *card* (cardinality) is the number of its discrete elements. If cardinality is not specified, its default value (currently, 256) is assumed. When limits are not explicitly defined, the universe of discourse is taken from 0 to 1.

The format of the linguistic label "*identifier*" is similar to the operator identifier, that is, "*package.function*" or simply "*function*" if the package where the user has defined the membership functions has been already imported.

XFL3 supports inheritance mechanisms in the type definitions (like its precursor, XFL). To express inheritance, the heading of the definition is as follows:

```
type identifier extends identifier {
```

The types so defined inherit automatically the universe of discourse and the labels of their parents. The labels defined in the body of the type are either added to the parent labels or overwrite them if they have the same name.

<pre>type Tinput1 [-90,90] { NM xfl.trapezoid(-100,-90,-40,-30); NP xfl.trapezoid(-40,-30,-10,0); CE xfl.triangle(-10,0,10); PP xfl.trapezoid(0,10,30,40); PM xfl.trapezoid(30,40,90,100); }</pre>	
--	--

<pre> type Tinput2 extends Tinput1 { NG xfl.trapezoid(-100,-90,-70,-60); NM xfl.trapezoid(-70,-60,-40,-30); PM xfl.trapezoid(30,40,60,70); PG xfl.trapezoid(60,70,90,100); } </pre>	
---	--

Rule bases

A rule base in XFL3 is an object containing the rules that define the logic relationships among the linguistic variables. Its definition format is as follows:

```

rulebase identifier (input_list : output_list) using operatorset {
  [factor] if (antecedent) -> consequent_list;
  [factor] if (antecedent) -> consequent_list;
  ..... }

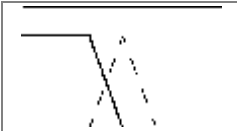
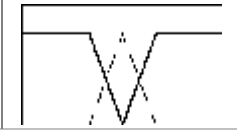



```

The definition format of the input and output variables is "*type identifier*", where type refers to one of the linguistic variable types previously defined. The operator set selection is optional, so that when it is not explicitly defined, the default operators are employed. Confidence weights or factors (with default values of 1) can be applied to the rules.

A rule antecedent describes the relationships among the input variables. XFL3 allows to express complex antecedents by combining basic propositions with connectives or linguistic hedges. On the other side, each rule consequent describes the assignation of a linguistic variable to an output variable as "*variable = label*".

A basic proposition relates an input variable with one of its linguistic labels. XFL3 admits several relationships, such as equality, inequality and several linguistic hedges. The following table shows the different relationships offered by XFL3.

Basic propositions	Description	Representation
variable == label	equal to	
variable >= label	equal or greater than	
variable <= label	equal or smaller than	
variable > label	greater than	

variable < label	smaller than	
variable != label	not equal to	
variable %= label	slightly equal to	
variable ~= label	moreorless equal to	
variable += label	strongly equal to	

In general, a rule antecedent is formed by a complex proposition. Complex propositions are composed of basic propositions, connected by fuzzy connectives and linguistic hedges. The following table shows how to generate complex propositions in XFL3.

Complex propositions	Description
proposition & proposition	and operator
proposition proposition	or operator
!proposition	not operator
%proposition	slightly operator
~proposition	moreorless operator
+proposition	strongly operator

This is an example of a rule base composed of some rules which include complex propositions.

```
rulebase basel(input1 x, input2 y : output z) using systemop {
  if( x == medium & y == medium) -> z = tall;
  [0.8] if( x <=short | y != very_tall ) -> z = short;
  if( +(x > tall) & (y ~= medium) ) -> z = tall;
  ..... }
```

System global behavior

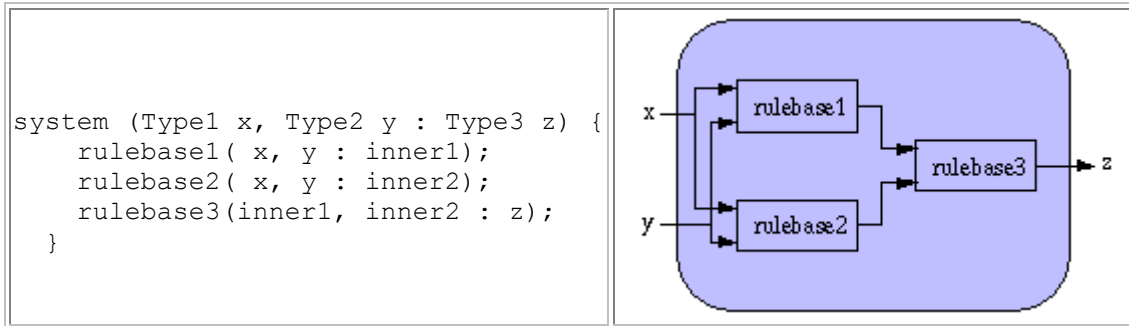
The description of the system global behavior means to define the global input and output variables of the system as well as the rule base hierarchy. This description is as follows in XFL3:

```

system (input_list : output_list) {
  rule_base_identifier(inputs : outputs);
  rule_base_identifier(inputs : outputs);
  ..... }

```

The definition format of the global input and output variables is the same format employed in the definition of the rule bases. The inner variables that may appear establish serial or parallel interconnections among the rule bases. Inner variables must firstly appear as output variables of a rule base before being employed as input variables of other rule bases.



Function packages

A great advantage of XFL3 is that functions assigned to fuzzy operators can be defined freely by the user in external files (named as packages), which gives a huge flexibility to the environment. Each package can include an unlimited number of definitions.

Four types of functions can be defined in XFL3: [binary functions](#) that can be used as T-norms, S-norms, and implication functions; [unary functions](#) that are related with linguistic hedges; [membership functions](#) that are used to describe linguistic labels; and [defuzzification methods](#).

A function definition include its name (and possible alias), the parameters that specify its behavior as well as the constraints on these parameters, the description of its behavior in the different languages to which it could be compiled (C, C++ and Java), and even the description of its differential function (if it is employed in gradient-based learning mechanisms). This information is the basis to generate automatically a Java class that incorporates all the function capabilities and can be employed by any XFL3 specification.

Binary function definition

Binary functions can be assigned to the conjunction operator (and), the disjunction operator (or), the implication function (imp), and the rule aggregation operator (also). The structure of a binary function definition in a function package is as follows:

```

binary identifier { blocks }

```

The blocks that can appear in a binary function definition are *alias*, *parameter*, *requires*, *java*, *ansi_c*, *cplusplus*, *derivative* and *source*.

The block *alias* is used to define alternative names to identify the function. Any of these identifiers can be used to refer the function. The syntax of the block *alias* is:

```
alias identifier, identifier, ... ;
```

The block *parameter* allows the definition of those parameters which the function depends on. Its format is:

```
parameter identifier, identifier, ... ;
```

The block *requires* expresses the constraints on the parameter values by means of a Java Boolean expression that validates the parameter values. The structure of this block is:

```
requires { expression }
```

The blocks *java*, *ansi_c* and *cplusplus* describe the function behavior by means of its description as a function body in Java, C and C++ programming languages, respectively. Input variables for these functions are 'a' and 'b'. The format of these blocks is the following:

```
java { Java_function_body }  
ansi_c { C_function_body }  
cplusplus { C++_function_body }
```

The block *derivative* describes the derivative function with respect to the input variables 'a' and 'b'. This description consists of a Java assignation expression to the variable '*deriv*[]'. The derivative function with respect to the input variable 'a' must be assigned to '*deriv*[0]', while the derivative function with respect to the input variable 'b' must be assigned to '*deriv*[1]'. The description of the derivative function allows to propagate the system error derivative used by the supervised learning algorithms based on gradient descent. The format is:

```
derivative { Java_expressions }
```

The block *source* is used to define Java code that is directly included in the class code generated for the function definition. This code allows to define local methods that can be used into other blocks. The structure is:

```
source { Java_code }
```

The following example shows the definition of the T-norm minimum, also used as Mamdani's implication function.

```

binary min {
  alias mamdani;
  java { return (a<b? a : b); }
  ansi_c { return (a<b? a : b); }
  cplusplus { return (a<b? a : b); }
  derivative {
    deriv[0] = (a<b? 1: (a==b? 0.5 : 0));
    deriv[1] = (a>b? 1: (a==b? 0.5 : 0));
  }
}

```

Unary function definition

Unary functions are used to describe the linguistic hedges. These functions can be assigned to the *not* modifier, the *very* or *strongly* modifier, the *more-or-less* modifier, and the *slightly* modifier. The structure of a unary function definition in a function package is as follows:

```
unary identifier { blocks }
```

The blocks that can appear in a unary function definition are *alias*, *parameter*, *requires*, *java*, *ansi_c*, *cplusplus*, *derivative* and *source*.

The block *alias* is used to define alternative names to identify the function. Any of these identifiers can be used to refer the function. The syntax of the block *alias* is:

```
alias identifier, identifier, ... ;
```

The block *parameter* allows the definition of those parameters which the function depends on. Its format is:

```
parameter identifier, identifier, ... ;
```

The block *requires* expresses the constraints on the parameter values by means of a Java Boolean expression that validates the parameter values. The structure of this block is:

```
requires { expression }
```

The blocks *java*, *ansi_c* and *cplusplus* describe the function behavior by means of its description as a function body in Java, C and C++ programming languages, respectively. Input variable for these functions is 'a'. The format of these blocks is the following:

```
java { Java_function_body }
ansi_c { C_function_body }
cplusplus { C++_function_body }
```

The block *derivative* describes the derivative function with respect to the input variable 'a'. This description consists of a Java assignation expression to the variable '*deriv*'. The description of the derivative function allows to propagate the

system error derivative used by the supervised learning algorithms based on gradient descent. The format is:

```
derivative { Java_expressions }
```

The block *source* is used to define Java code that is directly included in the class code generated for the function definition. This code allows to define local methods that can be used into other blocks. The structure is:

```
source { Java_code }
```

The following example shows the definition of the Yager C-norm, which depends on the parameter *w*.

```
unary yager {
  parameter w;
  requires { w>0 }
  java { return Math.pow( ( 1 - Math.pow(a,w) ) , 1/w ); }
  ansi_c { return pow( ( 1 - pow(a,w) ) , 1/w ); }
  cplusplus { return pow( ( 1 - pow(a,w) ) , 1/w ); }
  derivative { deriv = - Math.pow( Math.pow(a,-w) -1, (1-w)/w ); }
}
```

Membership function definition

The membership functions are assigned to the linguistic labels that form a linguistic variable type. The structure of a membership function definition in a function package is as follows:

```
mf identifier { blocks }
```

The blocks that can appear in a membership function definition are *alias*, *parameter*, *requires*, *java*, *ansi_c*, *cplusplus*, *derivative* and *source*.

The block *alias* is used to define alternative names to identify the function. Any of these identifiers can be used to refer the function. The syntax of the block *alias* is:

```
alias identifier, identifier, ... ;
```

The block *parameter* allows the definition of those parameters which the function depends on. Its format is:

```
parameter identifier, identifier, ... ;
```

The block *requires* expresses the constraints on the parameter values by means of a Java Boolean expression that validates the parameter values. This expression can also use the values of the variables '*min*' and '*max*', which represent the minimum and maximum values in the universe of discourse of the linguistic variable considered. The structure of this block is:

```
requires { expression }
```

The blocks *java*, *ansi_c* and *cplusplus* describe the function behavior by means of its description as a function body in Java, C and C++ programming languages, respectively. The format of these blocks is the following:

```
java { Java_function_body }  
ansi_c { C_function_body }  
cplusplus { C++_function_body }
```

The definition of a membership function includes not only the description of the function behavior itself, but also the function behavior under the greater-or-equal and smaller-or-equal modifications, and the computation of the center and basis values of the membership function. As a consequence, the blocks *java*, *ansi_c* and *cplusplus* are divided into the following subblocks:

```
equal { code }  
greatereq { code }  
smallereq { code }  
center { code }  
basis { code }
```

The subblock *equal* describes the function behavior. The subblocks *greatereq* and *smallereq* describe the greater-or-equal and smaller-or-equal modifications, respectively. The input variable in these subblocks is called 'x', and the code can use the values of the function parameters and the variables '*min*' and '*max*', which represent the minimum and maximum values of the universe of discourse of the function. The subblocks *greatereq* and *smallereq* can be omitted. In that case, these transformations are computed by sweeping all the values of the universe of discourse. However, it is much more efficient to use an analytical function, so that the definition of these subblocks is strongly recommended.

The subblocks *center* and *basis* describe the center and basis of the membership function. The code of these subblocks can use the values of the function parameters and the variables '*min*' and '*max*'. This information is used by several simplified defuzzification methods. These subblocks are optional and their default functions return a zero value.

The block *derivative* describes the derivative function with respect to each function parameter. This block is also divided into the subblocks *equal*, *greatereq*, *smallereq*, *center* and *basis*. The code of these subblocks consists of Java expressions assigning values to the variable '*deriv*[]'. The value of '*deriv*[i]' represents the derivative of each function with respect to the i-th parameter of the membership function. The description of the derivative function allows to compute the system error derivative used by gradient descent-based learning algorithms. The format is:

```
derivative { subblocks }
```

The block *source* is used to define Java code that is directly included in the class code generated for the function definition. This code allows to define local methods that can be used into other blocks. The structure is:

```
source { Java_code }
```

The following example shows the definition of the bell shaped membership function.

```
mf bell {
  parameter a, b;
  requires { a>=min && a<=max && b>0 }
  java {
    equal { return Math.exp( -(a-x)*(a-x)/(b*b) ); }
    greaterreq { if(x>a) return 1; return Math.exp( - (x-a)*(x-a)/(b*b)
); }
    smallereq { if(x<a) return 1; return Math.exp( - (x-a)*(x-a)/(b*b)
); }
    center { return a; }
    basis { return b; }
  }
  ansi_c {
    equal { return exp( -(a-x)*(a-x)/(b*b) ); }
    greaterreq { if(x>a) return 1; return exp( - (x-a)*(x-a)/(b*b) ); }
    smallereq { if(x<a) return 1; return exp( - (x-a)*(x-a)/(b*b) ); }
    center { return a; }
    basis { return b; }
  }
  cplusplus {
    equal { return exp( -(a-x)*(a-x)/(b*b) ); }
    greaterreq { if(x>a) return 1; return exp( - (x-a)*(x-a)/(b*b) ); }
    smallereq { if(x<a) return 1; return exp( - (x-a)*(x-a)/(b*b) ); }
    center { return a; }
    basis { return b; }
  }
  derivative {
    equal {
      double aux = (x-a)/b;
      deriv[0] = 2*aux*Math.exp(-aux*aux)/b;
      deriv[1] = 2*aux*aux*Math.exp(-aux*aux)/b;
    }
    greaterreq {
      if(x>a) { deriv[0] = 0; deriv[1] = 0; }
      else {
        double aux = (x-a)/b;
        deriv[0] = 2*aux*Math.exp(-aux*aux)/b;
        deriv[1] = 2*aux*aux*Math.exp(-aux*aux)/b;
      }
    }
    smallereq {
      if(x<a) { deriv[0] = 0; deriv[1] = 0; }
      else {
        double aux = (x-a)/b;
        deriv[0] = 2*aux*Math.exp(-aux*aux)/b;
        deriv[1] = 2*aux*aux*Math.exp(-aux*aux)/b;
      }
    }
    center { deriv[0] = 1; deriv[1] = 0; }
    basis { deriv[0] = 0; deriv[1] = 1; }
  }
}
```

Defuzzification method definition

Defuzzification methods obtain the representative value of a fuzzy set. These methods are used in the final stage of the fuzzy inference process, when it is not possible to work with fuzzy conclusions. The structure of a defuzzification method definition in a function package is as follows:

```
defuz identifier { blocks }
```

The blocks that can appear in a defuzzification method definition are *alias*, *parameter*, *requires*, *definedfor*, *java*, *ansi_c*, *cplusplus* and *source*.

The block *alias* is used to define alternative names to identify the method. Any of these identifiers can be used to refer the method. The syntax of the block *alias* is:

```
alias identifier, identifier, ... ;
```

The block *parameter* allows the definition of those parameters which the method depends on. Its format is:

```
parameter identifier, identifier, ... ;
```

The block *requires* expresses the constraints on the parameter values by means of a Java Boolean expression that validates the parameter values. The structure of this block is:

```
requires { expression }
```

The block *definedfor* is used to enumerate the types of membership functions that the method can use as partial conclusions. This block has been included because some simplified defuzzification methods only work with certain membership functions. This block is optional. By default, the method is assumed to work with all the membership functions. The structure of the block is:

```
definedfor identificador, identificador, ... ;
```

The block *source* is used to define Java code that is directly included in the class code generated for the method definition. This code allows to define local functions that can be used into other blocks. The structure is:

```
source { Java_code }
```

The blocks *java*, *ansi_c* and *cplusplus* describe the behavior of the method by means of its description as a function body in Java, C and C++ programming languages, respectively. The format of these blocks is the following:

```
java { Java_function_body }  
ansi_c { C_function_body }  
cplusplus { C++_function_body }
```

The input variable for these functions is the object *'mf'*, which encapsulates the fuzzy set obtained as the conclusion of the inference process. The code can use the

value of the variables '*min*', '*max*' and '*step*', which represent respectively the minimum, maximum and division of the universe of discourse of the fuzzy set. Conventional defuzzification methods are based on sweeps along all the values of the universe of discourse, and they compute the membership degree of each value in the universe. On the other side, simplified defuzzification methods use sweeps along the partial conclusions, and they compute the representative value in terms of the activation degree, center, basis and parameters of these partial conclusions. The way this information is accessed by the object *mf* depends on the programming language, as shown in the next table.

Description	java	ansi_c	cplusplus
membership degree	<i>mf.compute(x)</i>	<i>mf.compute(x)</i>	<i>mf.compute(x)</i>
partial conclusions	<i>mf.conc[]</i>	<i>mf.conc[]</i>	<i>mf.conc[]</i>
number of partial conclusions	<i>mf.conc.length</i>	<i>mf.length</i>	<i>mf.length</i>
activation degree of the i-th conclusion	<i>mf.conc[i].degree()</i>	<i>mf.degree[i]</i>	<i>mf.conc[i]->degree()</i>
center of the i-th conclusion	<i>mf.conc[i].center()</i>	<i>center(mf.conc[i])</i>	<i>mf.conc[i]->center()</i>
basis of the i-th conclusion	<i>mf.conc[i].basis()</i>	<i>basis(mf.conc[i])</i>	<i>mf.conc[i]->basis()</i>
j-th parameter of the i-th conclusion	<i>mf.conc[i].param(j)</i>	<i>param(mf.conc[i],j)</i>	<i>mf.conc[i]->param(j)</i>
number of the input variables in the rule base	<i>mf.input.length</i>	<i>mf.inputlength</i>	<i>mf.inputlength</i>
values of the input variables in the rule base	<i>mf.input[]</i>	<i>mf.input[]</i>	<i>mf.input[]</i>

The following example shows the definition of the classical CenterOfArea defuzzification method.

```
defuz CenterOfArea {
  alias CenterOfGravity, Centroid;
  java {
    double num=0, denom=0;
    for(double x=min; x<=max; x+=step) {
      double m = mf.compute(x);
      num += x*m;
      denom += m;
    }
    if(denom==0) return (min+max)/2;
    return num/denom;
  }
  ansi_c {
    double x, m, num=0, denom=0;
    for(x=min; x<=max; x+=step) {
      m = compute(mf,x);
      num += x*m;
      denom += m;
    }
  }
}
```

```

    if(denom==0) return (min+max)/2;
    return num/denom;
  }
  cplusplus {
    double num=0, denom=0;
    for(double x=min; x<=max; x+=step) {
      double m = mf.compute(x);
      num += x*m;
      denom += m;
    }
    if(denom==0) return (min+max)/2;
    return num/denom;
  }
}

```

The following example shows the definition of a simplified defuzzification method (Weighted Fuzzy Mean).

```

defuz WeightedFuzzyMean {
  definedfor triangle, isosceles, trapezoid, bell, rectangle;
  java {
    double num=0, denom=0;
    for(int i=0; i<mf.conc.length; i++) {
      num += mf.conc[i].degree()*mf.conc[i].basis()*mf.conc[i].center();
      denom += mf.conc[i].degree()*mf.conc[i].basis();
    }
    if(denom==0) return (min+max)/2;
    return num/denom;
  }
  ansi_c {
    double num=0, denom=0;
    int i;
    for(i=0; i<mf.length; i++) {
      num += mf.degree[i]*basis(mf.conc[i])*center(mf.conc[i]);
      denom += mf.degree[i]*basis(mf.conc[i]);
    }
    if(denom==0) return (min+max)/2;
    return num/denom;
  }
  cplusplus {
    double num=0, denom=0;
    for(int i=0; i<mf.length; i++) {
      num += mf.conc[i]->degree()*mf.conc[i]->basis()*mf.conc[i]-
>center();
      denom += mf.conc[i]->degree()*mf.conc[i]->basis();
    }
    if(denom==0) return (min+max)/2;
    return num/denom;
  }
}

```

This final example shows the definition of the 1st order Takagi-Sugeno method.

```

defuz TakagiSugeno {
  definedfor parametric;
  java {
    double denom=0;
    for(int i=0; i<mf.conc.length; i++) denom += mf.conc[i].degree();
    if(denom==0) return (min+max)/2;
    double num=0;
    for(int i=0; i<mf.conc.length; i++) {
      double f = mf.conc[i].param(0);
      for(int j=0; j<mf.input.length; j++) f +=
mf.conc[i].param(j+1)*mf.input[j];
      num += mf.conc[i].degree()*f;
    }
    return num/denom;
  }
  ansi_c {
    double f, num=0, denom=0;
    int i, j;
    for(i=0; i<mf.length; i++) denom += mf.degree[i];
    if(denom==0) return (min+max)/2;
    for(i=0; i<mf.length; i++) {
      f = param(mf.conc[i],0);
      for(j=0; j<mf.inputlength; j++) f +=
param(mf.conc[i],j+1)*mf.input[j];
      num += mf.degree[i]*f;
    }
    return num/denom;
  }
  cplusplus {
    double num=0, denom=0;
    for(int i=0; i<mf.length; i++) {
      double f = mf.conc[i]->param(0);
      for(int j=0; j<mf.inputlength; j++) f += mf.conc[i]-
>param(j+1)*mf.input[j];
      num += mf.conc[i]->degree()*f;
      denom += mf.conc[i]->degree();
    }
    if(denom==0) return (min+max)/2;
    return num/denom;
  }
}
}

```

The standard package *xfl*

The XFL3 specification language allows the user to define its own membership functions, defuzzification methods, and functions related with fuzzy connectives and linguistic hedges. In order to ease the use of XFL3, the most well-known functions have been included in a standard package called *xfl*. The binary functions included are the following:

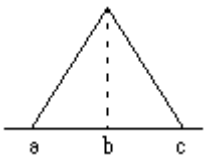
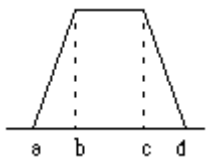
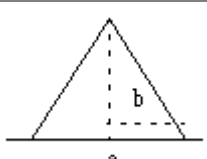
Name	Type	Java description
min	T-norm	$(a < b ? a : b)$
prod	T-norm	$(a * b)$
bounded_prod	T-norm	$(a + b - 1 > 0 ? a + b - 1 : 0)$

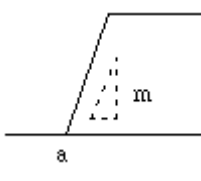
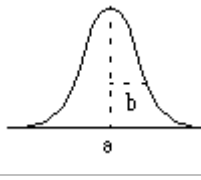
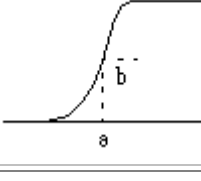
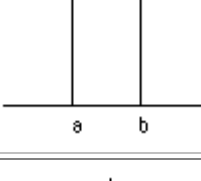
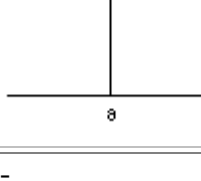
drastic_prod	T-norm	$(a==1? b: (b==1? a : 0))$
max	S-norm	$(a>b? a : b)$
sum	S-norm	$(a+b-a*b)$
bounded_sum	S-norm	$(a+b<1? a+b: 1)$
drastic_sum	S-norm	$(a==0? b : (b==0? a : 0))$
dienes_resher	Implication	$(b>1-a? b : 1-a)$
mizumoto	Implication	$(1-a+a*b)$
lukasiewicz	Implication	$(b<a? 1-a+b : 1)$
dubois_prade	Implication	$(b==0? 1-a : (a==1? b : 1))$
zadeh	Implication	$(a<0.5 1-a>b? 1-a : (a<b? a : b))$
goguen	Implication	$(a<b? 1 : b/a)$
godel	Implication	$(a<=b? 1 : b)$
sharp	Implication	$(a<=b? 1 : 0)$

The unary functions included in the package *xfl* are:

Name	Parameter	Java description
not	-	$(1-a)$
sugeno	l	$(1-a)/(1+a*l)$
yager	w	$\text{Math.pow}((1 - \text{Math.pow}(a,w)) , 1/w)$
pow	w	$\text{Math.pow}(a,w)$
parabola	-	$4*a*(1-a)$

The membership functions defined in the package *xfl* are the following:

Name	Parameters	Description
triangle	a,b,c	
trapezoid	a,b,c,d	
isosceles	a,b	

slope	a,m	
bell	a,b	
sigma	a,b	
rectangle	a,b	
singleton	a	
parametric	unlimited	-

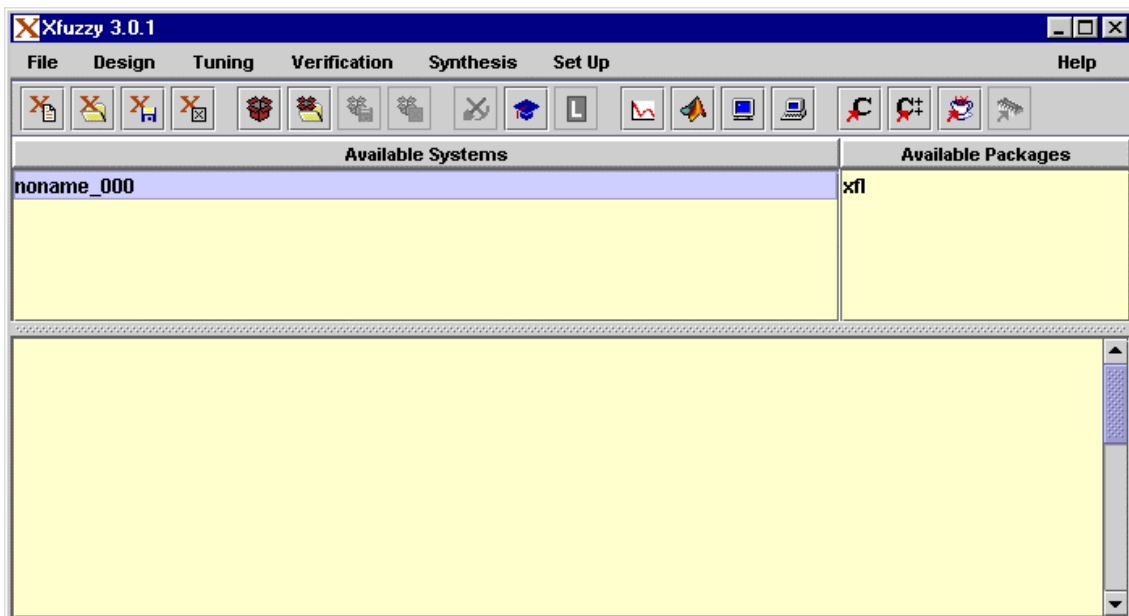
The defuzzification methods defined in the standard package are:

Name	Type	Defined for
CenterOfArea	Conventional	any function
FirstOfMaxima	Conventional	any function
LastOfMaxima	Conventional	any function
MeanOfMaxima	Conventional	any function
FuzzyMean	Simplified	triangle, isosceles, trapezoid, bell, rectangle, singleton
WeightedFuzzyMean	Simplified	triangle, isosceles, trapezoid, bell, rectangle
Quality	Simplified	triangle, isosceles, trapezoid, bell, rectangle
GammaQuality	Simplified	triangle, isosceles, trapezoid, bell, rectangle
MaxLabel	Simplified	singleton
TakagiSugeno	Simplified	parametric

The Xfuzzy 3.0 development environment

- [The Xfuzzy 3.0 development environment](#)
 - [Description stage](#)
 - [System edition \(xfedit\)](#)
 - [Package edition \(xfpkg\)](#)
 - [Verification stage](#)
 - [2-dimensional representation \(xf2dplot\)](#)
 - [3-dimensional representation \(xf3dplot\)](#)
 - [Inference monitor \(xfmt\)](#)
 - [System simulation \(xfsim\)](#)
 - [Tuning stage](#)
 - [Supervised learning \(xfsl\)](#)
 - [Synthesis stage](#)
 - [C code generator \(xfc\)](#)
 - [C++ code generator \(xfcpp\)](#)
 - [Java code generator \(xfj\)](#)

Xfuzzy 3.0 is a development environment for designing fuzzy systems, which integrates several tools covering the different stages of the design. The environment integrates all these tools under a graphical user interface which eases the design process. The next figure shows the main window of the environment.



The menu bar in the main window contains the links to the different tools. Under the menu bar, there is a button bar with the most used options. The central zone of the window shows two lists. The first one is the list of loaded systems (the environment can work with several systems simultaneously). The second list

contains the loaded packages. The rest of the main window is occupied by a message area.

The menu bar is divided into the different stages of the system development. The *File* menu allows to create, load, save and close a fuzzy system. This menu contains also the options to create, load, save and close a function package. The menu ends with the option to exit the environment. The *Design* menu is used to edit a selected fuzzy system ([xfedit](#)) or a selected package ([xfpkg](#)). The *Tuning* menu contains the links to the knowledge acquisition tool (under development), the supervised learning tool ([xfsl](#)), and the reinforcement learning tool (under development). The *Verification* menu allows to represent the system behavior on a 2-dimensional plot ([xf2dplot](#)) or a 3-dimensional plot ([xf3dplot](#)), monitoring the system ([xfmt](#)), and simulating it ([xfsim](#)). The *Synthesis* menu is divided into two parts: the software synthesis, that generates system descriptions in C ([xfc](#)), C++ ([xfcpp](#)), and Java ([xfj](#)); and the hardware synthesis, that implements a system description as a fuzzy circuit (under development). The *Set Up* menu is used to modify the environment working directory, to save the environment messages in an external log file, to close the log file, to clean up the message area of the main window, and to change the look and feel of the environment.

Many options on the menu bar are only enabled when a fuzzy system is selected. A fuzzy system is selected by just clicking its name in the system list. Double clicking the name will open the edition tool. The same result is obtained by pressing the *Enter* key once the system has been selected. The *Insert* key will create a new system and the *Delete* key is used to close the system. These shortcuts are common to all the lists of the environment: the *Insert* key is used to insert a new element on a list; the *Enter* key or a double click will edit the selected element; and the *Delete* key will remove the element from the list.

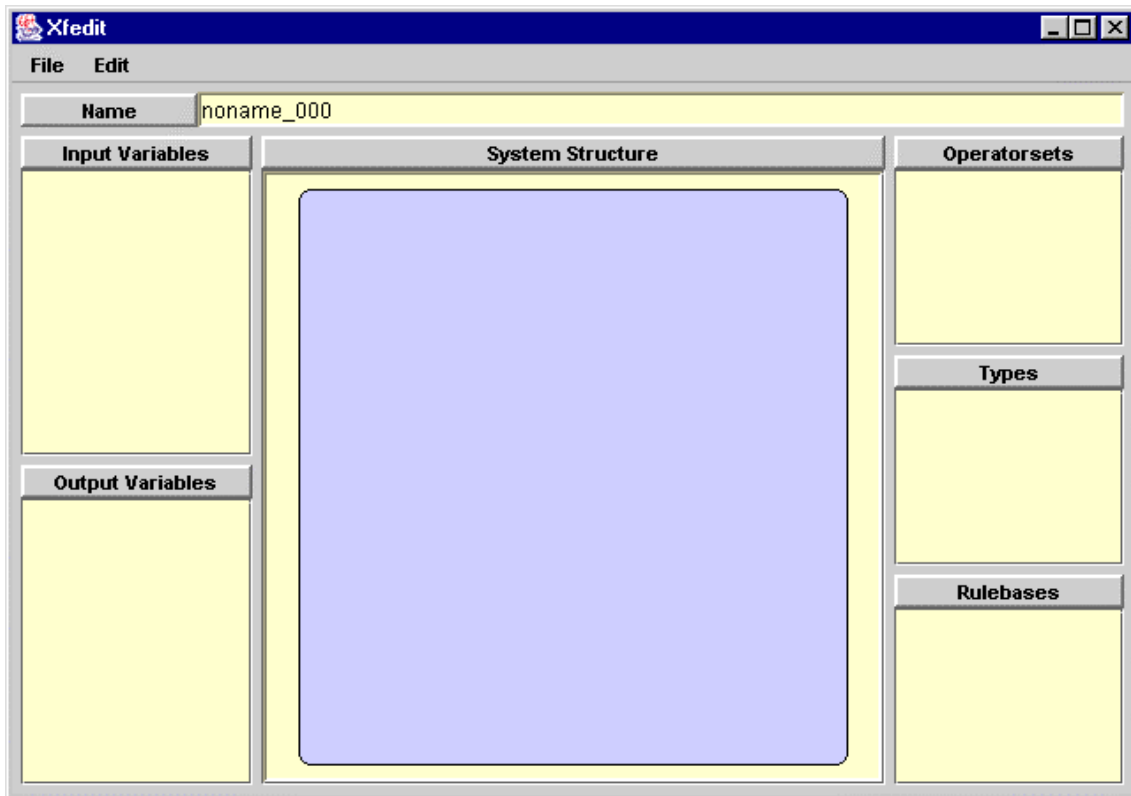
Description stage

The first step in the development of a fuzzy system is to select a preliminary description of the system. This description will be later refined as a result of the tuning and verification stages.

Xfuzzy 3.0 contains two tools assisting in the description of fuzzy systems: [xfedit](#) and [xfpkg](#). The first one is dedicated to the logical definition of the system, that is, the definition of its linguistic variables and the logical relations between them. On the other side, the [xfpkg](#) tool eases the description of the mathematical functions assigned to the fuzzy operators, linguistic hedges, membership functions and defuzzification methods.

The system edition tool - Xfedit

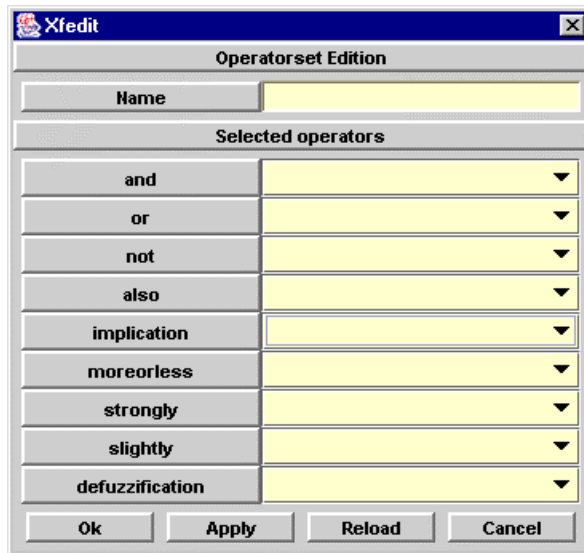
The *xfedit* tool offers a graphical interface to ease the description of fuzzy systems, avoiding the need for an in depth knowledge of the XFL3 language. The tool is formed by a set of windows that allows the user to create and edit the operator sets, linguistic variable types, and rule bases included in the fuzzy system, as well as describing the hierarchical structure of the system under development. The tool can be executed directly from the command line with the expression "*xfedit file.xfl*", or from the environment's main window, using the *System Edition* option in the *Design* menu.



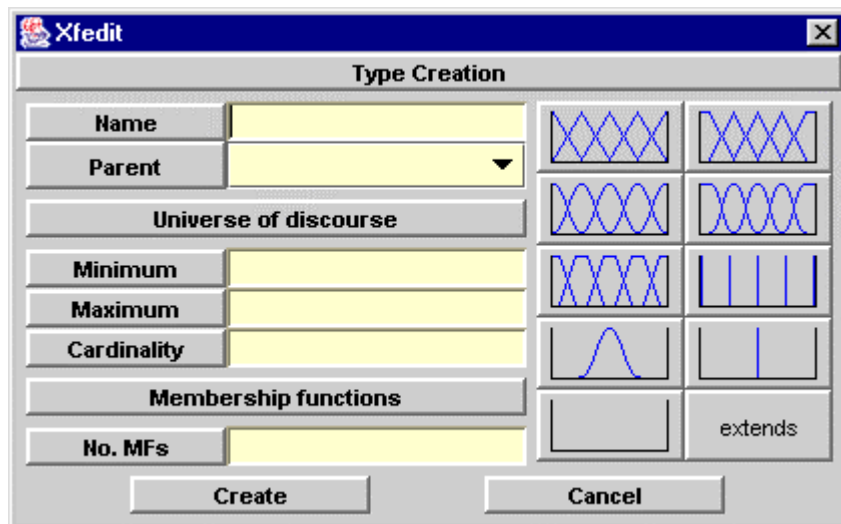
The figure shows the main window of *xfedit*. The *File* menu contains the following options: "Save", "Save As", "Load Package", "Edit XFL3 File" and "Close Edition". The options "Save" and "Save As" are used to save the present state of the system definition. The option "Load Package" is used to import new functions that can be assigned to the different fuzzy operators. The XFL3 file edition option opens a text window to edit the XFL3 description of the system. The last option is used to close the tool. The field *Name* under the menu bar is not editable. The name of the system under development can be changed by the *Save As* option. The body of the window is divided into three parts: the left one contains the lists of input and output global variables; the right part includes the lists of the defined operator sets, linguistic variable types and rule bases; finally, the central zone shows the hierarchical structure of the system.

The shortcuts of the different lists are the common ones of the environment: the *Insert* key creates a new element for each list; the *Delete* key is used to remove an element (when it has not been used); and the *Enter* key or a double click allows the element edition.

The creation of a fuzzy system in Xfuzzy usually starts with the definition of the [operator sets](#). The figure shows the window for editing operator sets in *xfedit*. It has a simple behavior. The first field contains the identifier of the operator set. The remaining fields contain pulldown lists to assign functions to the different fuzzy operators. If the selected function needs the introduction of some parameters, a new window will ask for them. The functions available in each list are those defined in the loaded packages. It is not necessary to make a choice for every field. At the bottom of the window, a command bar presents four options: "Ok", "Apply", "Reload" and "Cancel". The first option saves the operator set and closes the window. The second one just saves the last changes. The third option actualizes the field with the last saved values. The last one closes the window rejecting the last changes.

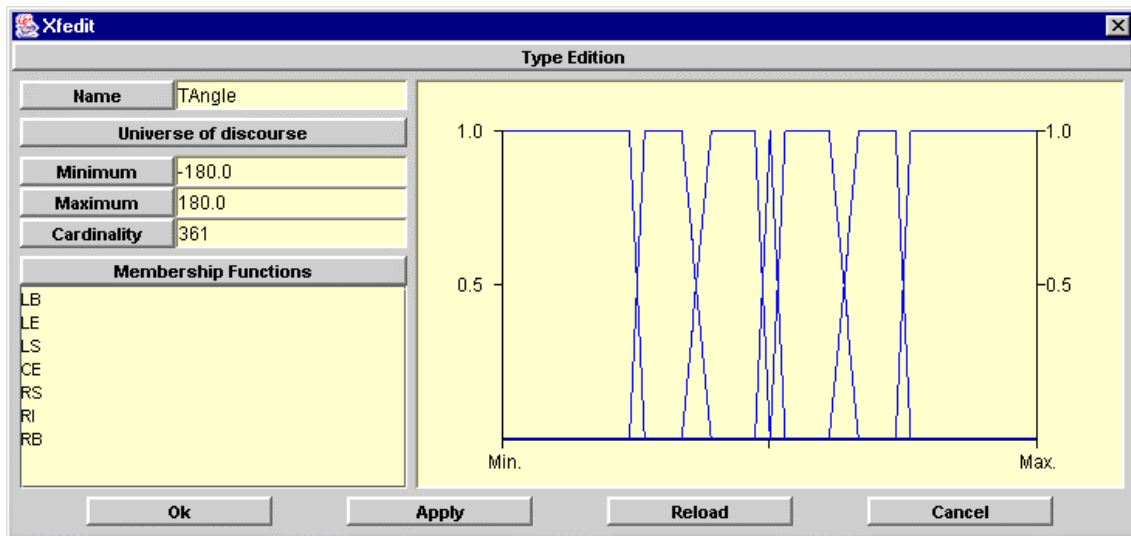


The following step in the description of a fuzzy system is to create the [linguistic variable types](#), by means of the *Type Creation* window shown below. A new type needs the introduction of its identifier and universe of discourse (minimum, maximum and cardinality). The window includes several predefined types corresponding to the most usual partitions of the universes. These predefined types contain homogeneous triangular, trapezoidal, bell-shaped and singleton partitions, shouldered-triangular and shouldered-bell partitions. Other predefined types are equal bells and singletons, which are commonly used as a first option for output variable types. When one of the previous predefined types is selected, the number of membership function of the partition must be introduced. The predefined types also include a blank option, which generates a type without any membership function, and the extension of an existing type (selected in the *Parent* field), that implements the inheritance mechanism of XFL3.

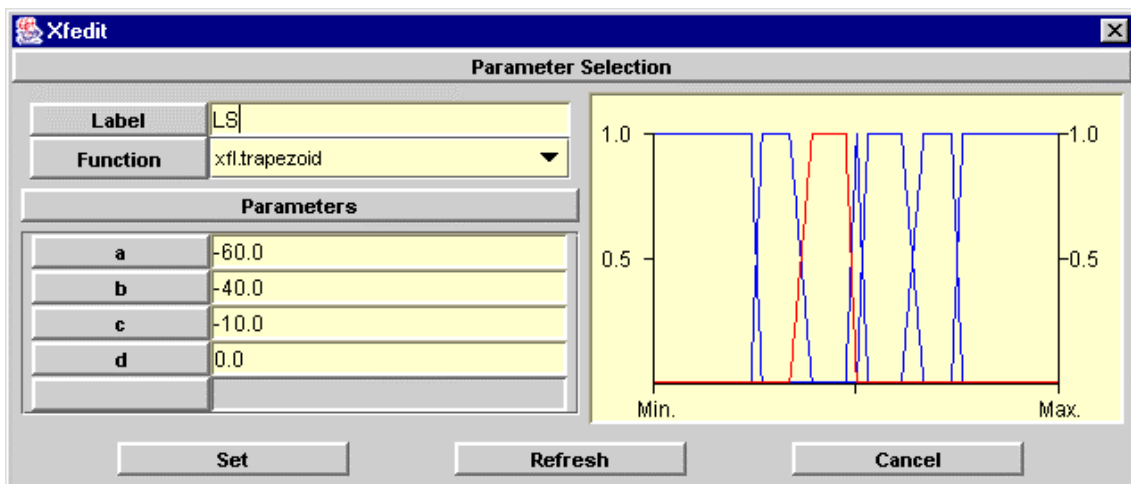


Once a type has been created, it can be edited using the *Type Edition* window. This window allows the modification of the type name and universe of discourse, for instance by adding, editing and removing the membership functions of the edited type. The window shows a graphical representation of the membership functions, where the selected membership function is represented in a different color. The bottom of the window presents a command bar with the usual buttons to save or reject the last changes, and to close the window. It is worth considering that the

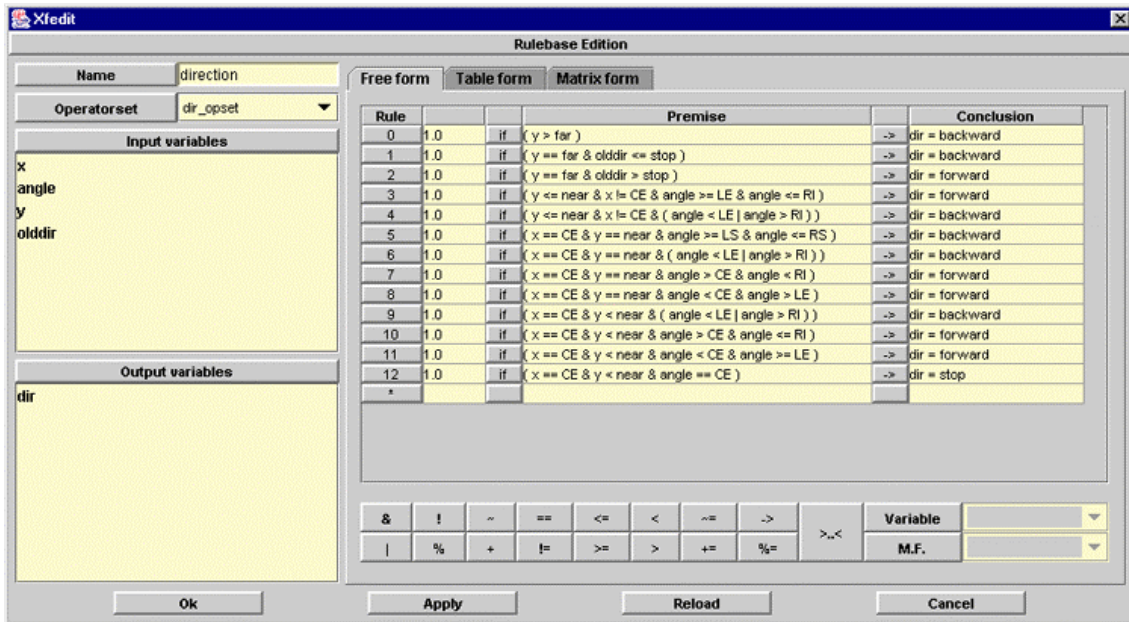
modifications on the definition of the universe of discourse can affect the membership functions. Hence, a validation of the membership function parameters is done before saving the modifications, and an error message appear whenever a membership function definition becomes invalid.



A membership function can be created or edited from the MF list with the usual accelerators (*Insert* key and *Enter* key or double click). The previous figure shows the window for editing a membership function. The window has fields to introduce the name of the linguistic label, to select the kind of membership function, and to introduce the parameter values. The right side of the window shows a graphical representation of all the membership functions, with the function being edited shown in a different color. The bottom of the window shows a command bar with three options: *Set*, to close the window saving the changes, *Refresh*, to repaint the graphical representation, and *Cancel*, to close the window without saving the modifications.



The third step in the definition of a fuzzy system is to describe the [rule bases](#) expressing the relationship among the system variables. Rule bases can be created, edited and removed from their list with the usual shortcuts (*Insert* key, *Enter* key or double click, and *Delete* key). The following window eases the edition of the rule bases.

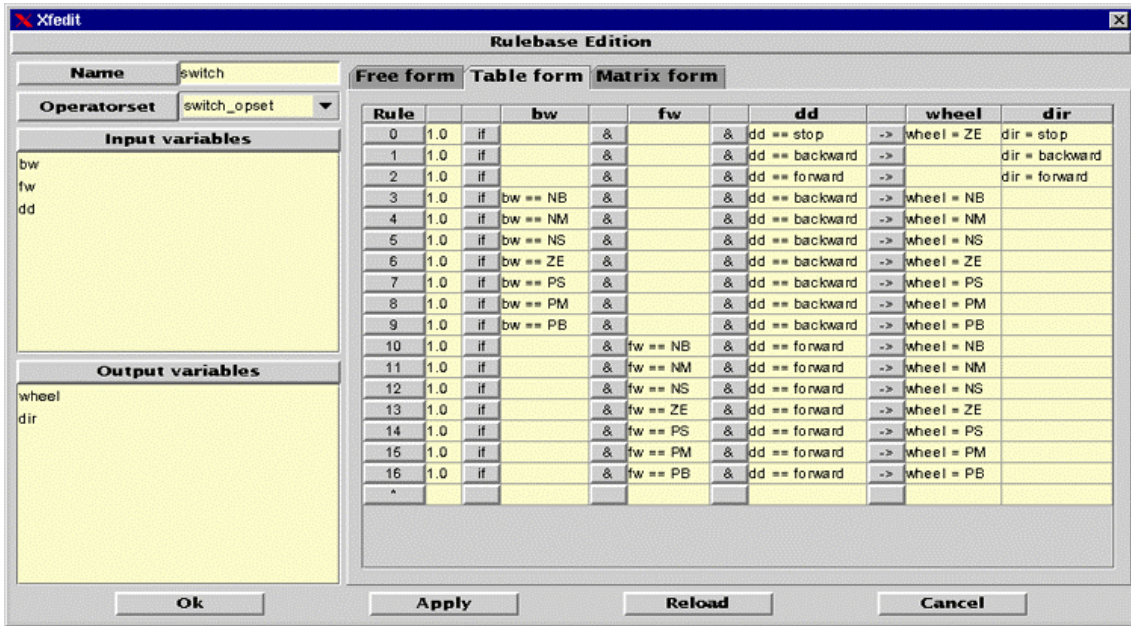


The rule base edition window is divided into three zones: the left side has the fields to introduce the names of the rule base and the operator set used, and to introduce the lists of input and output variables; the right zone is dedicated to showing the contents of the rules included in the rule base; and the bottom part of the window contains the command bar with the usual buttons to save or reject the modifications, and to close the window.

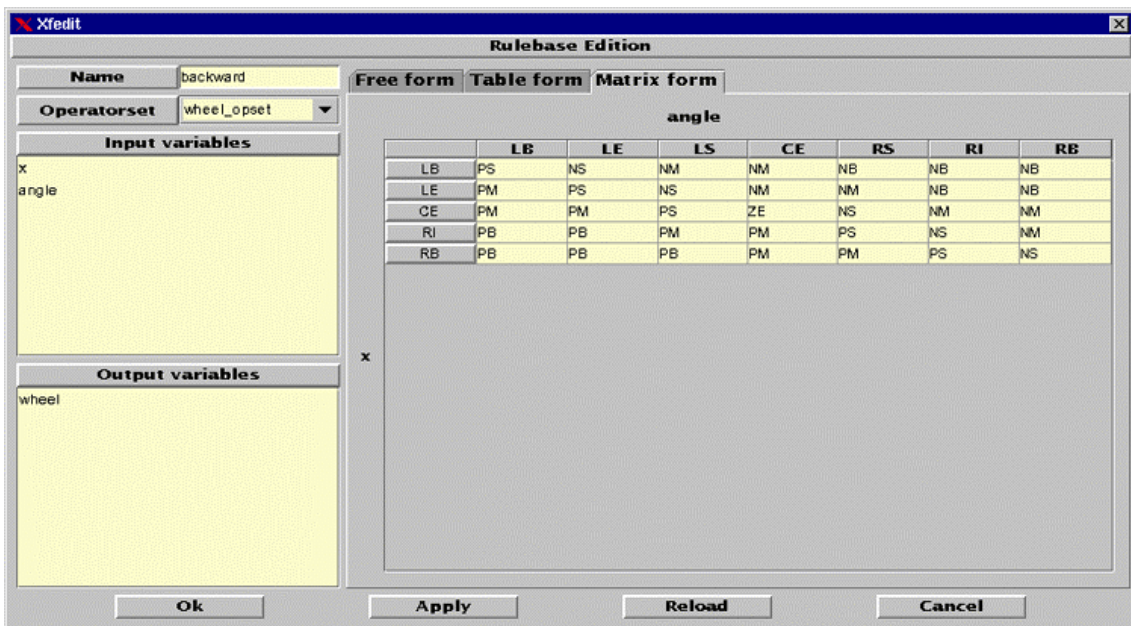
The input and output variables can be created, edited, or removed with the common list bindkeys. The information required by a variable definition is the name and the type of the variable.

The contents of the rules can be displayed in three formats: free, tabular, and matricial. The free format uses three fields for each rule. The first one contains the confidence weight. The second field shows the antecedent of the rule. This is an auto-editable field, where changes can be made by selecting the term to modify (a "?" symbol means a blank term) and by using the buttons of the window. The third field of each rule contains the consequent description. This is also an auto-editable field that can be modified by clicking the "->" button. New rules can be generated by introducing values on the last row (marked with the "*" symbol).

The button bar at the bottom of the free form allows to create conjunction terms ("&" button), disjunction terms ("|" button), modified terms with the linguistic hedges *not* ("!" button), *more or less* ("~" button), *slightly* ("% " button), and *strongly* ("+" button), and single terms relating a variable and a label with the clauses *equal to* ("=="), *not equal to* ("!="), *greater than* (">"), *smaller than* ("<"), *greater or equal to* (">="), *smaller or equal to* ("<="), *approximately equal to* ("~="), *strongly equal to* ("+="), and *slightly equal to* ("%="). The "->" button is used to add a rule conclusion. The ">..<" button is used to remove a conjunction or disjunction term (e.g. a term "v == l & ?" is transformed into "v == l"). The free form allows the user to describe more complex relationships among the variables than the other forms.

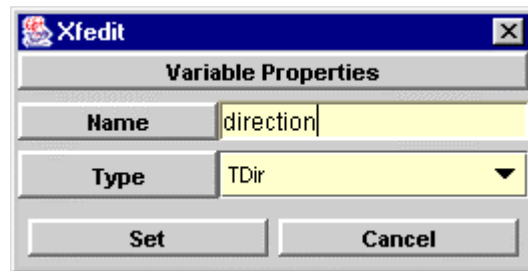


The tabular format is useful to define rules whose antecedent use only the operators *and* and *equal*. Each rule has a field to introduce the confidence weight and a pulldown list per input and output variables. There is no need of selecting all the variables fields, but one input and one output variables have always to be selected. If a rule base contains a rule that cannot be expressed in the tabular format, the table form can not be opened and an error message appears instead.

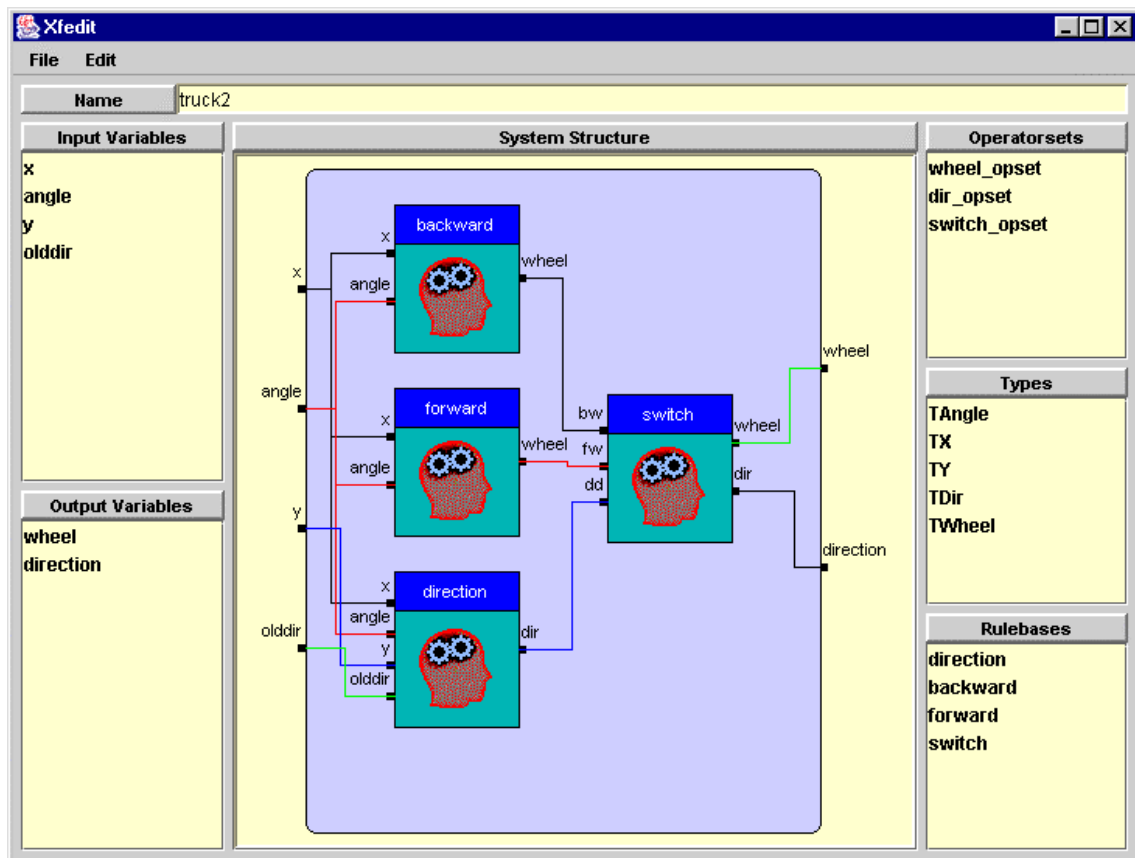


The matricial format is specially designed to describe a 2-input 1-output rule base. This form shows the content of a rule base in a clear and compact way. The matrix form generates rules such as "if(x==X & y==Y) -> z=Z", i.e., rules with a 1.0 confidence weight and formed by the conjunction of two equalities. Those rule bases that do not have the proper number of variables, or that contain rules with a different format, can not be shown in this form.

Once the operator sets, variable types, and rule bases have been defined; the following step in a fuzzy system definition is to define the global input and output variables by using the *Variable Properties* window. The information required to create a variable is its name and type.



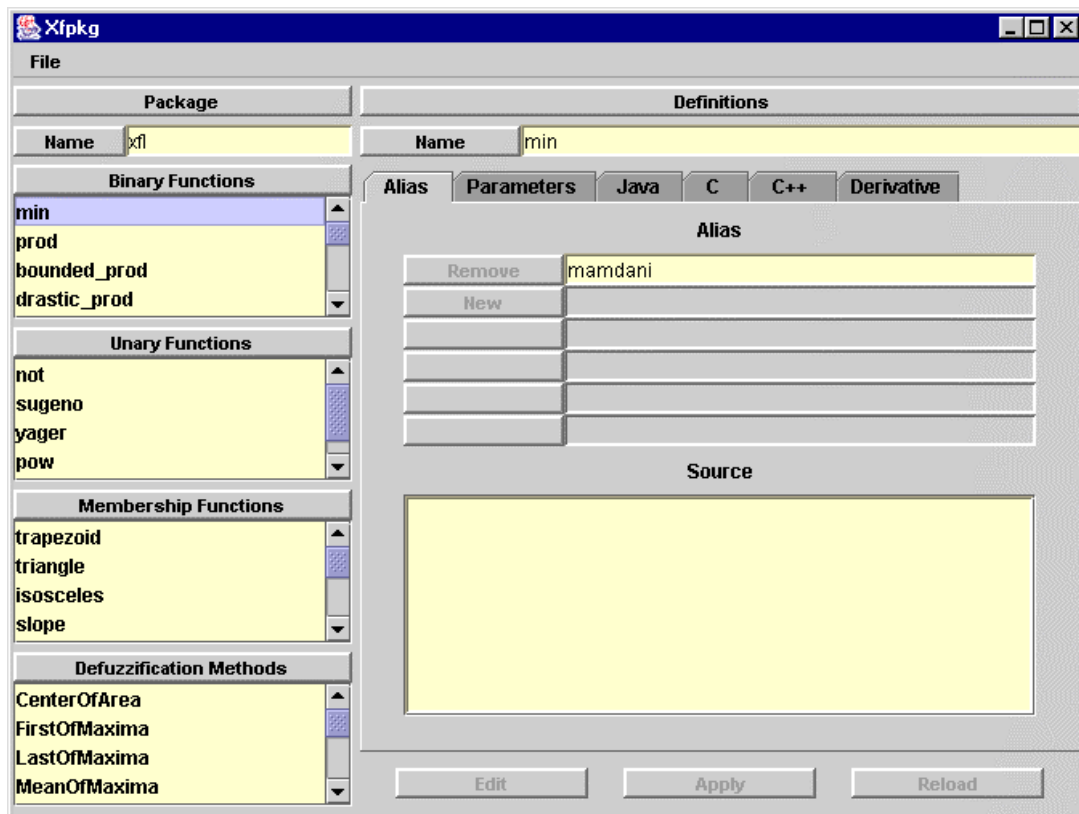
The final step in a fuzzy system definition is the description of its (possibly hierarchical) structure. The bindkey used to introduce a new module (a call to a rule base) in a hierarchy is the *Insert* key. To make links between the modules, the user must press the mouse over the node representing the origin variable and release the button over the destination variable node. To remove a link, the user must be select it by clicking on the destination variable node, and then press the *Delete* key. The tool does not allow to create a loop between modules.



The package edition tool - Xfpkg

The description of a fuzzy system within the Xfuzzy 3.0 environment is divided into two parts. The system logical structure (including the definitions of operator sets, variable types, rule bases, and hierarchical behavior structure) is specified in files with the extension ".xfl", and can be graphically edited with [xfedit](#). On the other hand, the mathematical description of the functions used as fuzzy connectives, linguistic hedges, membership functions, and defuzzification methods are specified in [packages](#).

The *xfpkg* tool is dedicated to easing the package edition. The tool implements a graphical user interface that shows the list of the different functions included in the package, and the contents of the different fields of a function definition. Most of these fields contains code describing the function in different programming languages. This code must be introduced manually. The tool can be executed from the command line or from the main window of the environment, using the option *Edit package* in the *Design* menu.



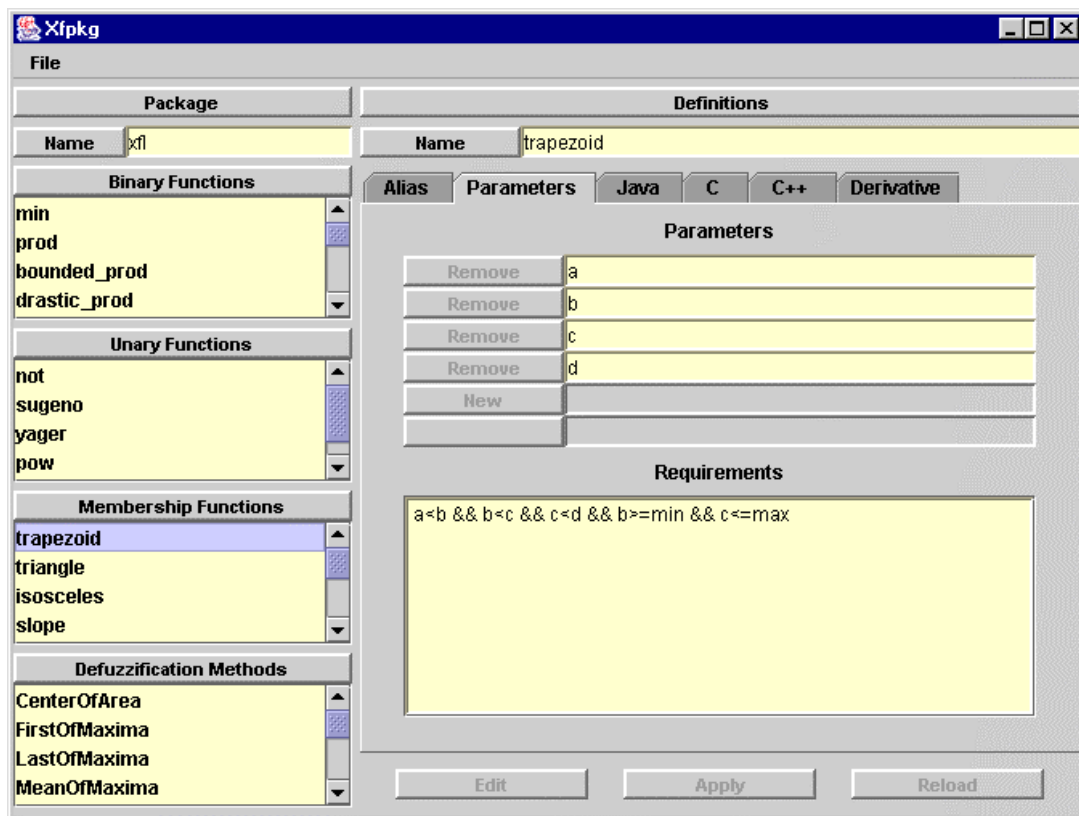
The previous figure shows the main window of *xfpkg*. The *File* menu contains the options "Save", "Save as", "Compile", "Delete" and "Close edition". The first two options are used to save the package file. The option "Compile" carries out a compilation process that generates the ".java" and ".class" files related to each function defined in the package. The option "Delete" is used to remove the package file and all the ".java" and ".class" files generated by the compilation process. The last option is used to close the tool.

The main window is divided into two parts. The left zone contains four lists showing the different kinds of functions included in the package: binary functions (related to conjunction, disjunction, aggregation, and implication operators), unary functions (associated with linguistic hedges), membership functions (related to linguistic

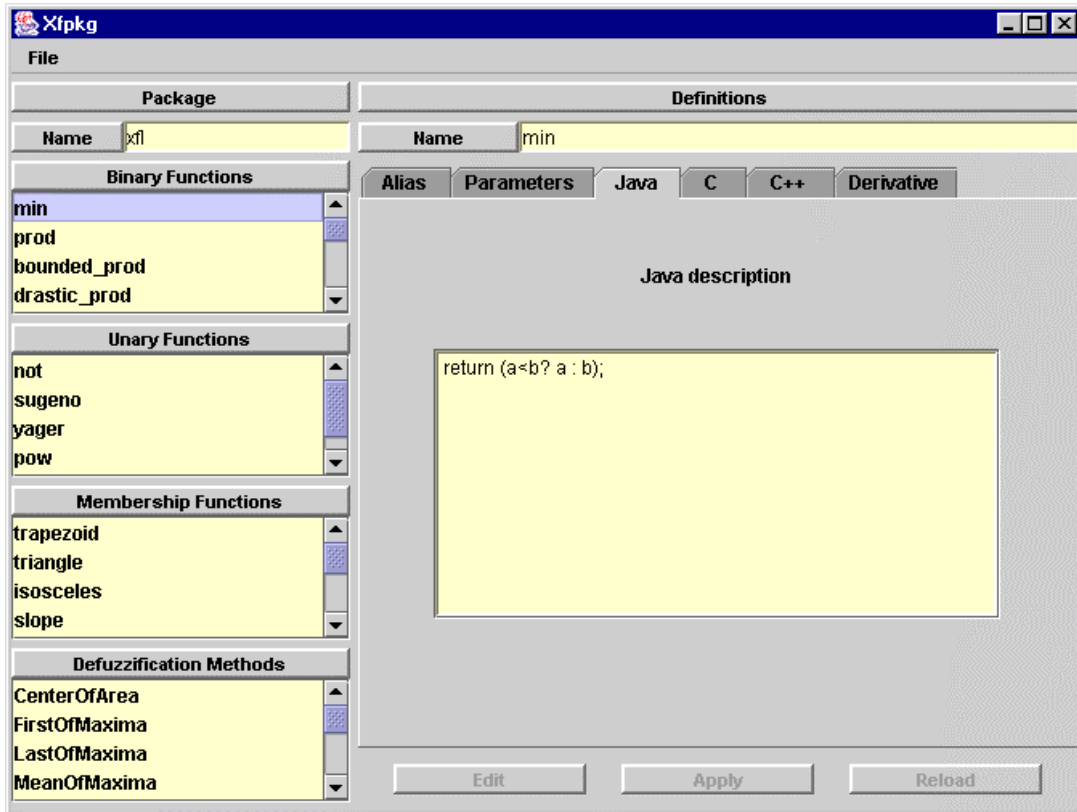
labels), and defuzzification methods (used to obtain representative values of the fuzzy conclusions). The right part of the main window shows the content of the different fields of a function definition. The bottom of this part contains a group of three buttons: "Edit", "Apply" and "Reload". When a function is selected in a list, its fields cannot be modified at first. The *Edit* command is used to allow the user modifying the fields. The *Apply* command saves the changes of the definition. This includes the generation of the ".java" and ".class" files. The *Reload* command rejects the modifications and actualizes the fields with the previous values.

The fields of a function definition are distributed among six tabbed panels. The *Alias* panel contains the list of alternative identifiers and the source block with the Java code of local methods that can be used in another fields and that are directly incorporated in the ".java" file.

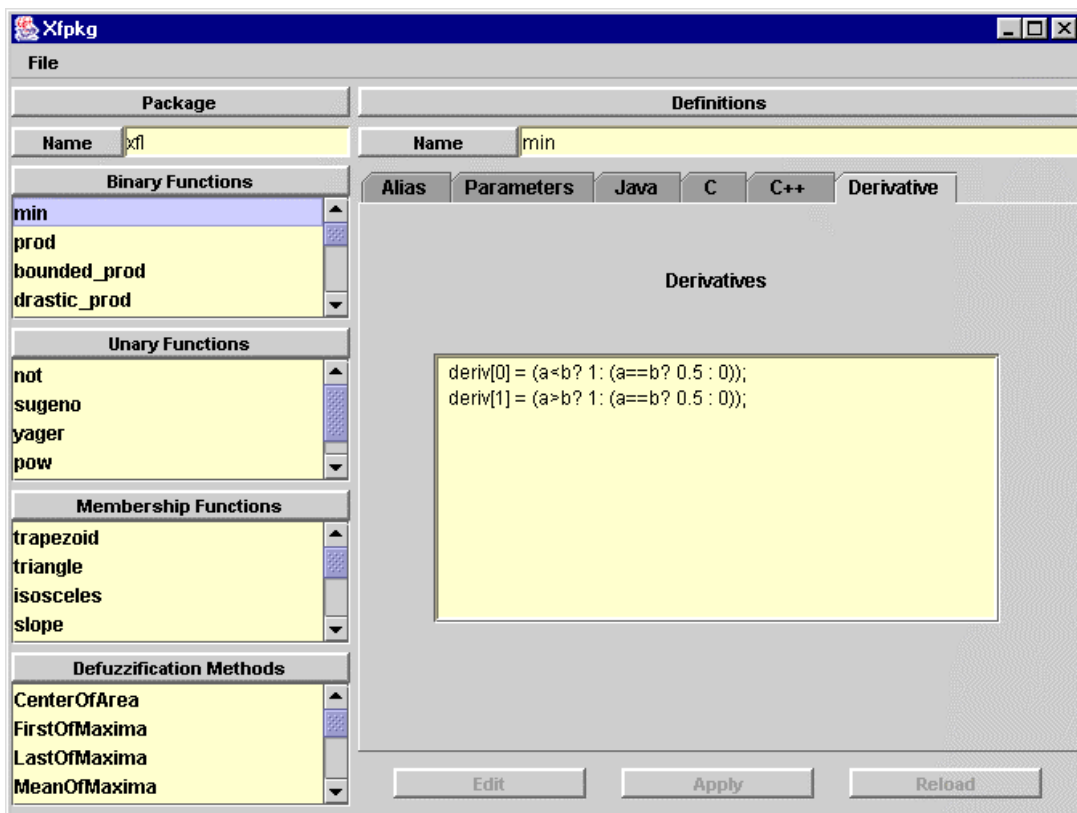
The *Parameters* panel contains the enumeration of the parameters used by the edited function. The panel also includes the field *requires*, to describe the constraints on the parameter values.



The *Java*, *C* and *C++* panels contain the description of the function behavior in these programming languages.

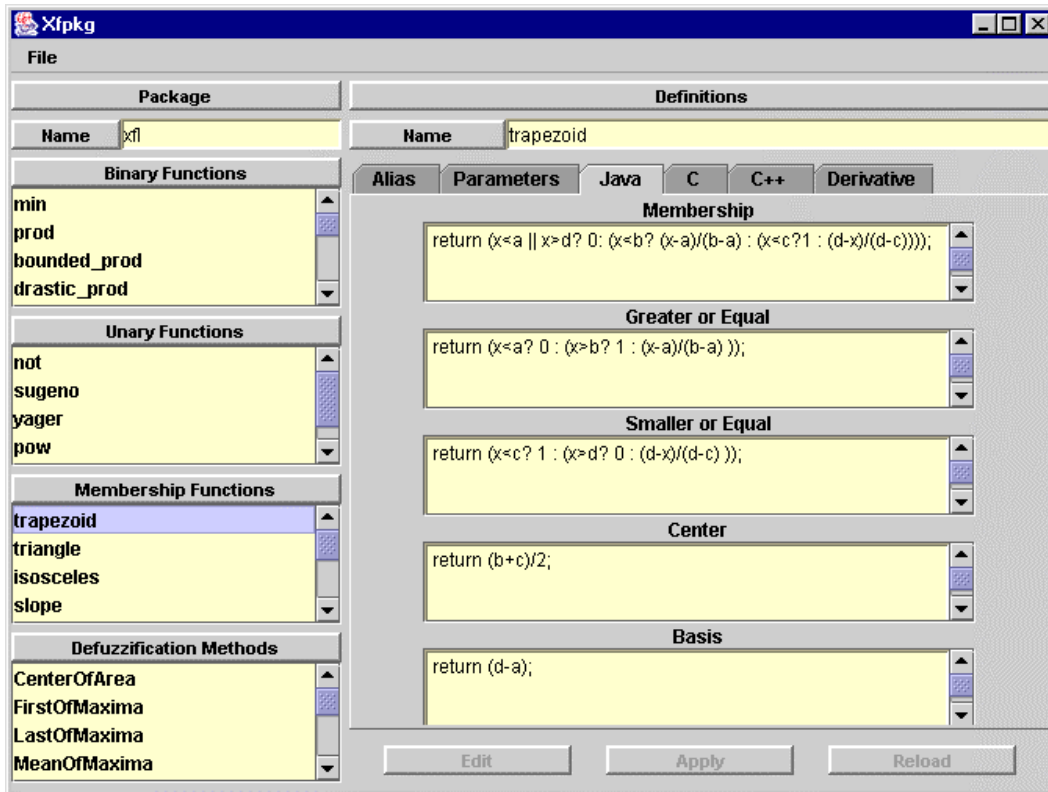


The last panel contains the description of the derivative function.

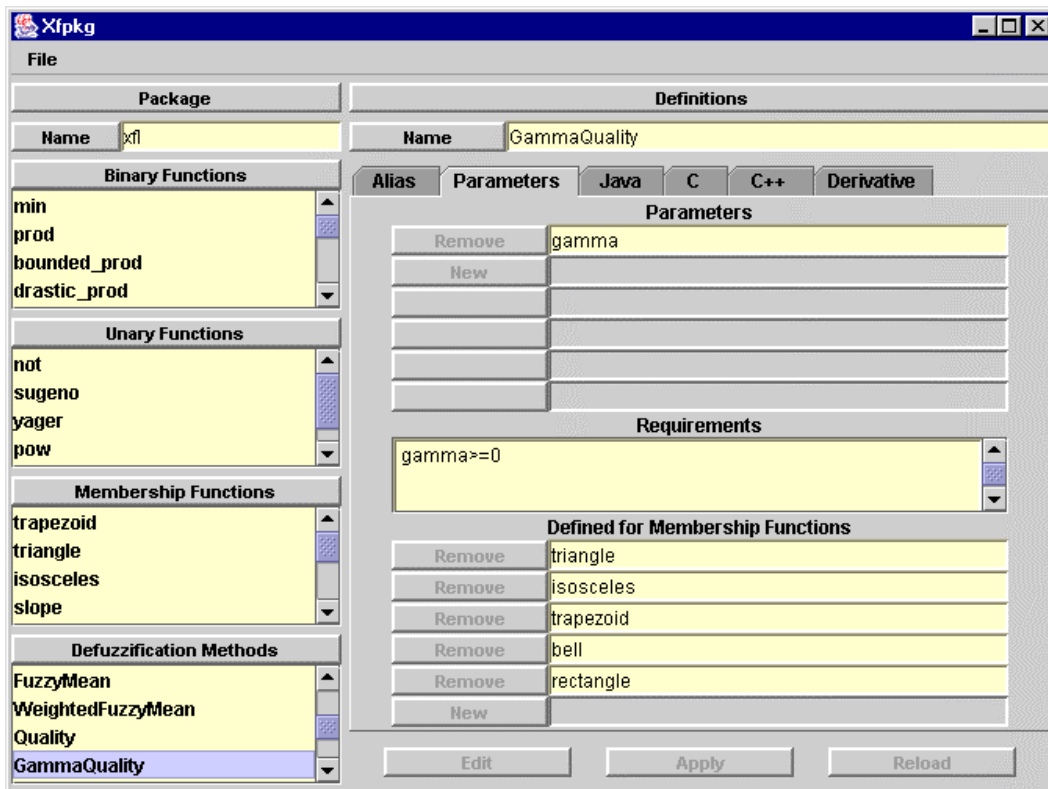


The membership function definition requires additional information to describe the function behavior in the different programming languages. In these cases, the *Java*,

C, C++ and *Derivative* panels contain four fields to show the contents of the subblocks *equal*, *greatereq*, *smallereq*, *center*, and *basis*.



Regarding defuzzification methods, they can include the enumeration of the membership functions that can be used by each method. This enumeration appears in the *Parameters* panel.



The *xfpkg* tool implements a graphical interface that allows the user to view and edit the definition of the functions included into a package file. This tool is used to describe the mathematical behavior of the defined functions in a graphical way. So, this tool is the complement of the *xfedit* tool, which describes the logical structure of the system, in the fuzzy system description stage.

Verification stage

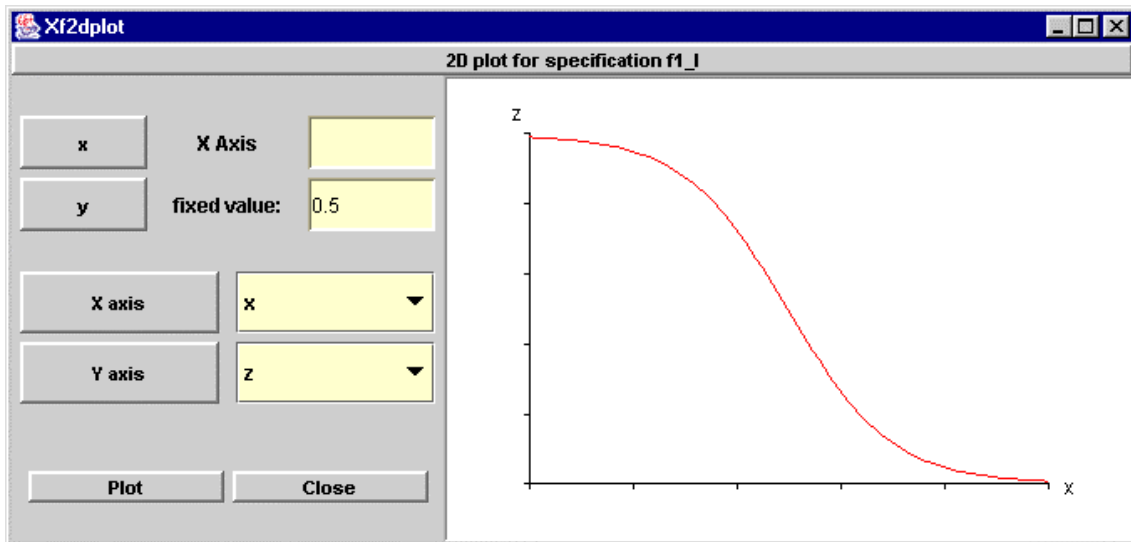
The verification stage in the fuzzy system design process consists in studying the behavior of the fuzzy system under development. The aim of this stage is the detection of probable deviations on the expected behavior and the identification of the sources of these deviations.

The Xfuzzy 3.0 environment covers the verification stage with four tools. The first one is [xf2dplot](#), which shows the system behavior by a two-dimensional plot. The second tool, [xf3dplot](#), implements a three-dimensional graphical representation of the system behavior. The monitor tool, [xfmt](#), shows the activation degree of every linguistic label and logical rule, as well as the value of the different inner variables, for a given set of input values. The last tool, [xfsim](#), is aimed at simulating the system within its actual or modeled operational environment. It allows illustrating the system evolution by means of a graphical representation of user-selected variables.

The 2-dimensional graphical representation tool - Xf2dplot

The *xf2dplot* tool allows studying the behavior of an output variable of the fuzzy system as a function of an input variable. The tool implements a 2-dimensional graphical representation that shows the variation of the selected output variable with respect to the selected input variable. When the system contains more than one input variable, the user must introduce a value to each non-selected input variable. The tool can be executed from the command line with the expression "*xf2dplot file.xfl*", or from the main window of the environment, using the option "*2D Plot*" in the *Verification* menu.

The main window of the tool is divided into two parts: the left one is devoted to configuring the graphical representation, while the right part is occupied by the plot. The configuration zone is formed by a set of fields dedicated to the introduction of the fixed values of the non-selected input variables. Two pulldown lists allow the selection of the input and output variable that are going to be represented. Finally, two buttons in the bottom of the configuration zone are used to actualize the graphical representation (*Plot*) and exits the tool (*Close*).

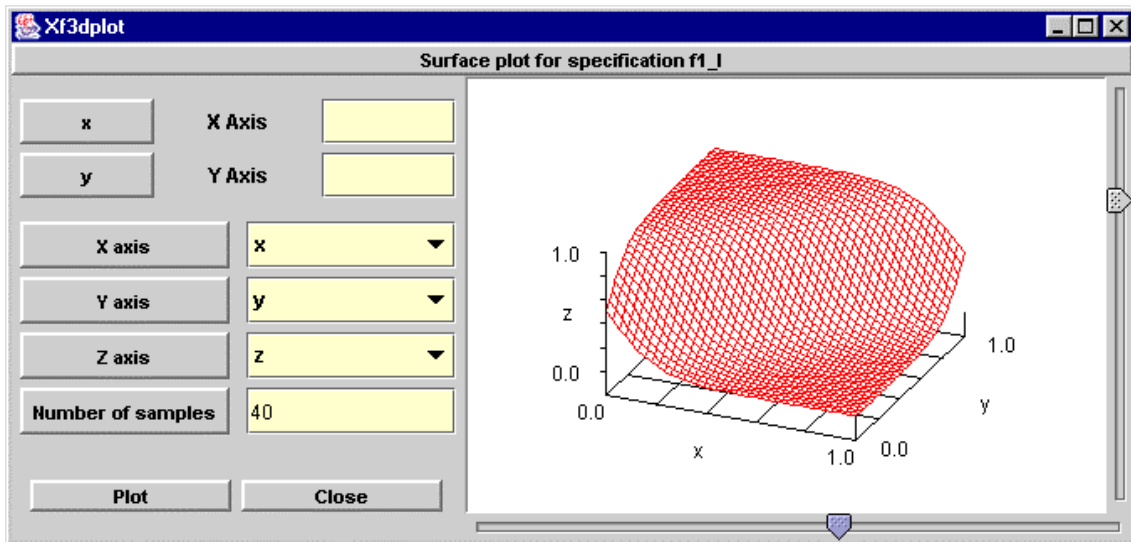


The 3-dimensional graphical representation tool - Xf3dplot

The *xf3dplot* tool illustrates the behavior of a fuzzy system by a 3-dimensional representation, i.e., a surface plot showing an output variable as a function of two input variables. Hence, the system to be represented must contain at least two input variables. If the system contains more than two input variables, the non-selected ones have to be specified by the user. The tool can be executed from the command line with the expression "*xf3dplot file.xfl*", or from the main window of the environment, using the option "*Surface 3D Plot*" in the *Verification* menu.

The main window of the tool is divided into two parts: the left one is dedicated to configuring the graphical representation, while the right part of the window is occupied by the surface plot. The configuration zone is formed by a set of fields dedicated to introducing the fixed values of the non-selected input variables. Three pulldown lists allow the selection of the variables assigned to each axis. The last field contains the number of points used in the partition of the X and Y axis. This is an important parameter because it determines the representation resolution. A low value in this parameter can exclude important details of the system behavior. On the other hand, a high value will make it difficult to understand the represented surface, as it will use a very dense grid. The default value of this parameter is 40. The configuration zone ends with a couple of buttons. The first one (*Plot*) is used to actualize the graphical representation with the present configuration. The second one (*Close*) exits the tool.

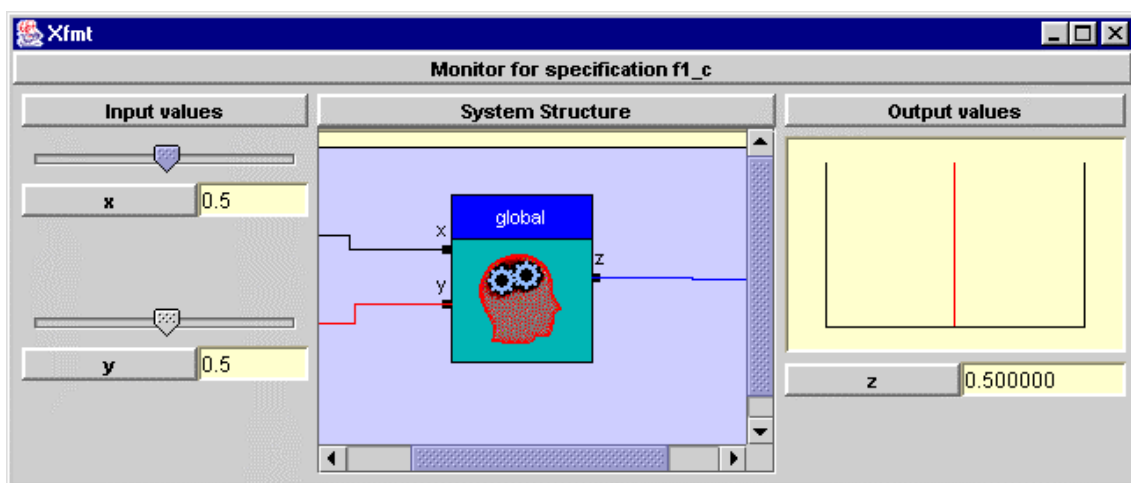
The graphical representation includes the possibility of rotating the surface by using two sliding buttons at the right and bottom parts of the plot. This rotation capability eases the interpretation of the represented surface.



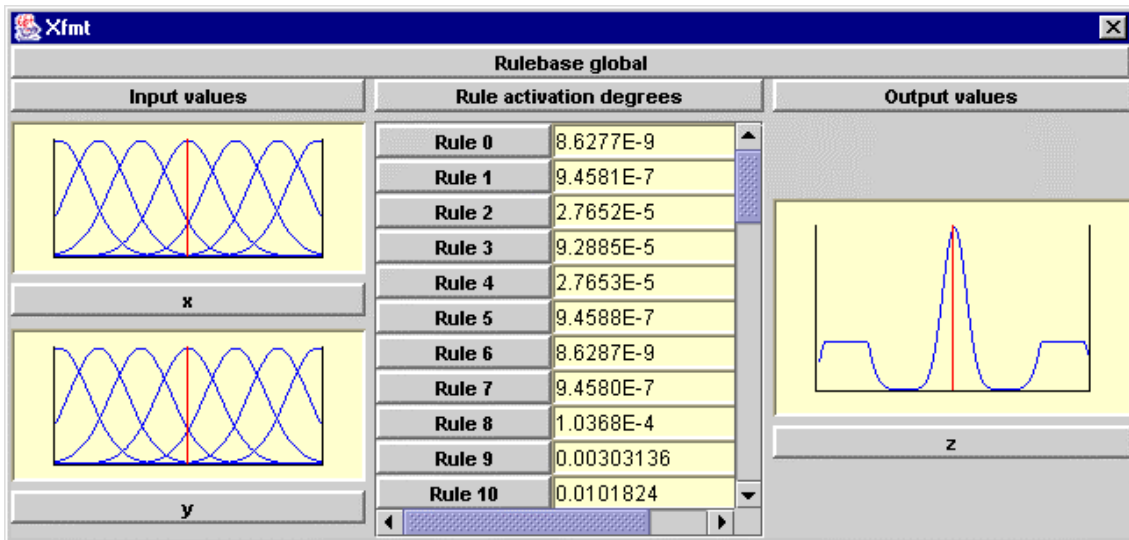
The inference monitor tool - Xfmt

The aim of the *xfmt* tool is to monitor the fuzzy inference process in the system, i.e., to show graphically the values of the different inner variables and the activation degree of the logical rules and linguistic labels, for a given set of input values. The tool can be executed from the command line with the expression "*xfmt file.xfl*", or from the main window of the environment, using the option "Monitor" in the *Verification* menu.

The main window of *xfmt* is divided into three parts. The left zone is used to introduce the values of the global input variables. For each variable, there is a field to introduce manually its value, and a sliding button to introduce the value as a position within the variable range. The right side shows the fuzzy set associated with the value of the global output variables, as well as the crisp (defuzzified) value for that variable. This crisp value is also shown as a singleton in the plot of the fuzzy set (if the fuzzy set is already a singleton, this plot only shows this singleton). The center of the window illustrates the (hierarchical) structure of the system.



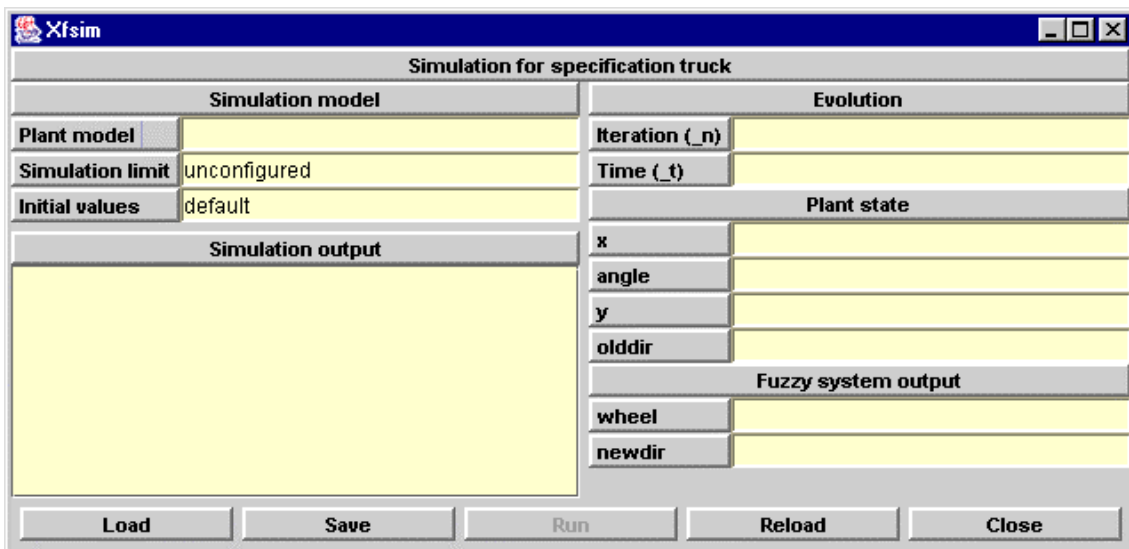
The tool also includes a window to monitor the inner values of the inference process on each rule base. To open this window, just click on the rule base on the hierarchical structure of the system.



The rule base monitor window is divided into three parts. The values of the input variables are shown at the left as singleton values within the membership functions assigned to the different linguistic labels. The center of the window contains a set of fields with the activation degree of each rule. The right side shows the values of the output variables obtained by the inference process. If the operator set used in the rule base specifies a defuzzification method, the output value is defuzzified, and the variable plot shows not only the fuzzy value but also the crisp value that is finally assigned to the output variable.

The simulation tool - Xfsim

The *xfsim* tool is dedicated to study feedback systems. The tool implements a simulation of the system behavior connected to the plant. The tool can be executed from the command line with the expression "*xfsim file.xfl*", or from the main window of the environment with the option "*Simulation*" in the *Verification* menu.



The main window of *xfsim* is shown in the figure. The configuration of the simulation process is made at the left side of the window, while the right side shows the status of the feedback system. The bottom of the window contains a menu bar with the options "*Load*", "*Save*", "*Run/Stop*", "*Reload*" and "*Close*". The first option

is used to load a configuration for the simulation process. The second one saves the present configuration on an external file. The *Run/Stop* option is used to start and stop the simulation process. The *Reload* option rejects the current simulation and reinitializes the tool. The last option exits the tool.

The configuration of a simulation process is done by the selection of the plant model connected with the fuzzy system and the description of the plant initial values, the end conditions, and a list of desired outputs for the simulation process. These outputs can be a log file to save the values of some selected variables, and graphical representations of these variables. The simulation status contains the number of iterations, the elapsed time for the initialization of the simulation, the values of the fuzzy system input variables, which represent the plant status, and the values of the fuzzy system output variables, which represent the action of the fuzzy system on the plant.

The plant connected to the fuzzy system is described by a file with '.class' extension, containing the Java binary code of a class describing the plant behavior. This class must implement the interface *xfuzzy.PlantModel* whose code is the following:

```
package xfuzzy;

public interface PlantModel {
    public void init() throws Exception;
    public void init(double[] state) throws Exception;
    public double[] state();
    public double[] compute(double[] x);
}
```

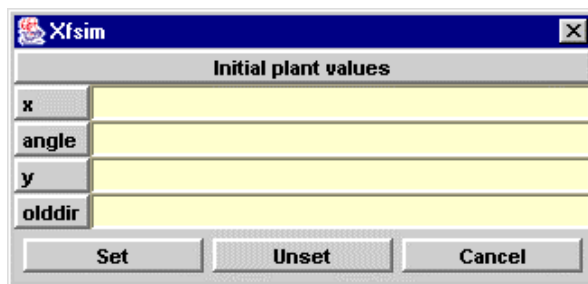
The function *init()* is used to initialize the plant with its default values, and must generate an exception when these values are not defined or cannot be assigned to the plant. The function *init(double[])* is used to set the initial values of the plant status to the selected values. It also generates an exception when these values cannot be assigned to the plant. The function *state()* returns the values of the plant status, which correspond to the input variables of the fuzzy system. Finally, the function *compute(double[])* modifies the plant status in terms of the fuzzy system output values. The user must write and compile this class on his own.

Defining a plant by a Java class offers a great flexibility to describe external systems. The simplest way consists in describing a mathematical model of the evolution of the plant from its state and the output values of the fuzzy system. In this scheme, the functions *init* and *state* assign and return, respectively, the values of the inner status variables, while the *compute* function implements the mathematical model. A more complex scheme consists in using a real plant connected to the computer (usually by a data acquisition board). In this case, the function *init* must initialize the data acquisition system, the function *state* must capture the current state of the plant, and the function *compute* must write the action on the data acquisition board as well as capture the new status of the plant.

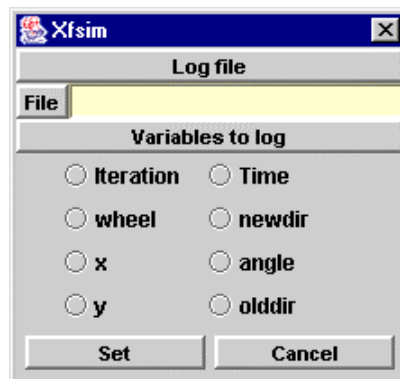
The configuration of the simulation process also requires the introduction of some end conditions. The window for selecting them contains a set of fields with the limit values of the simulation state variables.



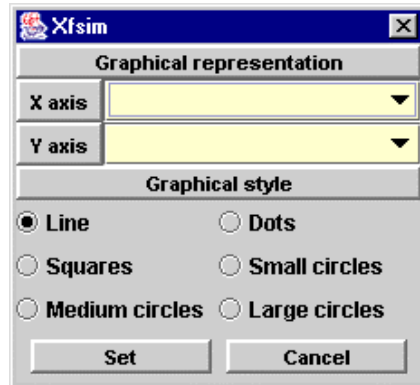
The initial state of the plant is described by using the following window. It contains a set of fields related to the plant variables, which correspond to the fuzzy system input variables.



The *xfsim* tool can provide graphical representations of the simulation process, as well as, saving the simulation results into a log file. The *Insert* key is used to introduce a new representation, as usual. This will open a window asking for the type of representation: either a log file, or a graphical plot. The window for describing a log file has a field to select the name of the file, and some buttons to choose the variables to be saved.



The window for describing the graphical representation contains two pulldown lists to select the variable assigned to the X and Y axis, and a set of buttons to choose the representation style.



The configuration of a simulation process can be saved to an external file, and loaded from a previously saved file. The contents of this file is composed by the following directives:

```
xfsim_plant("filename")
xfsim_init(value, value, ...)
xfsim_limit(limit & limit & ...)
xfsim_log("filename", varname, varname, ...)
xfsim_plot(varname, varname, style)
```

The directive *xfsim_plant* contains the file name of the Java binary code file describing the plant. The directive *xfsim_init* contains the value of the initial state of the plant. If this directive does not appear in the configuration file, the default initial state is assumed. The directive *xfsim_limit* contains the definition of the end conditions, which are expressed as a set of limits separated by the character &. The format of each limit is "variable < value" for the upper limits, and "variable > value" for the lower limits. The log files are described in the directive *xfsim_log*, which includes the name of the log file and the list of the variables to be saved. The graphical representations are defined by the directive *xfsim_plot*, which includes the names of the variables assigned to the X and Y axis, and the representation style. A style value of 0 means a plot with lines; value 1 indicates a dotted plot; value 2 makes the plot to use squares; and values 3, 4 and 5 indicate the use of circles of different sizes.

The next figure shows an example of a Java class implementing the plant model of a vehicle. This is the plant model that can be connected to the fuzzy system *truck* included in the environment examples. The state of the vehicle is stored in the inner variable *state[]*. The functions *init* just assign the values to the state components: the first component is the X position; the second one is the angle; the third one is the Y position and the last one contains the variable *olddir*. These components correspond to the input variables of the fuzzy system. The function *state* returns the inner variable. The vehicle dynamics is described by the function *compute*. The inputs to this function are the output variables of the fuzzy system. So, *val[0]* contains the value of the variable *wheel*, and *val[1]* contains the value of *direction*. The turn of the wheels leads to a change in the angle, and the new angle and direction allows the computation of the new position of the vehicle.

```

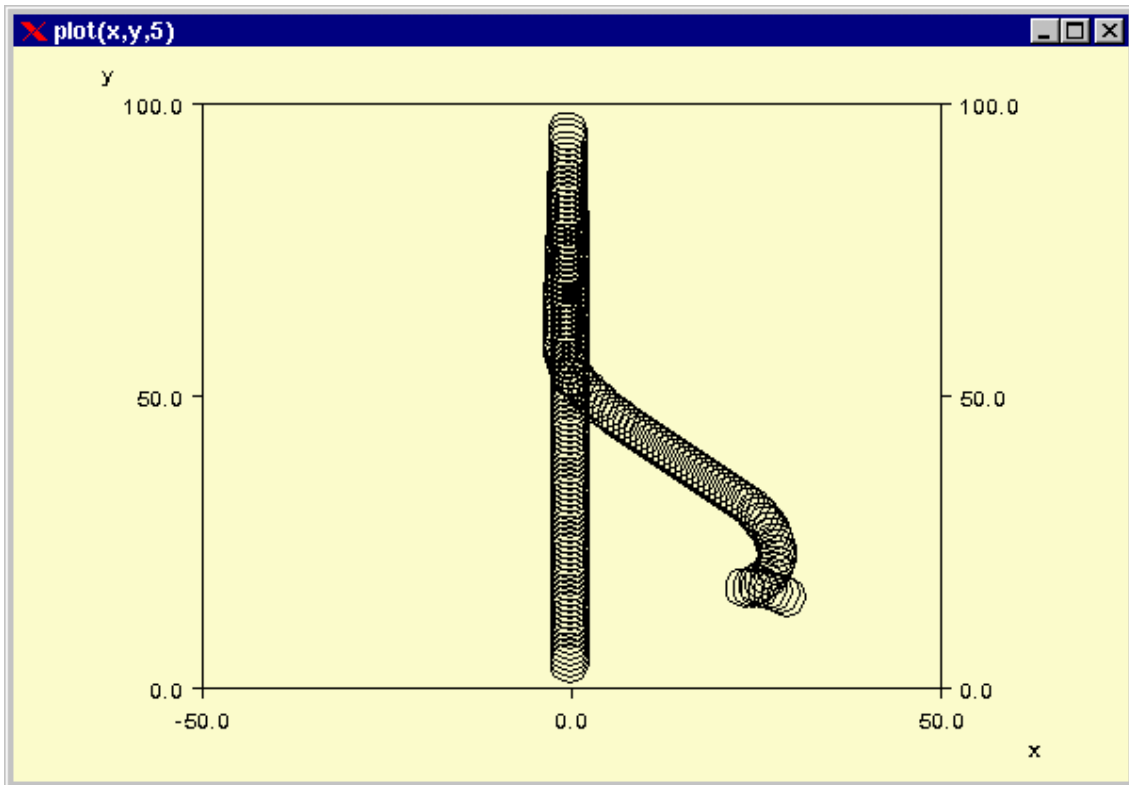
public class TruckModel implements xfuzzy.PlantModel {
    private double state[];

    public TruckModel {
        state = new double[4];
    }
    public void init() {
        state[0] = 0;
        state[1] = 0;
        state[2] = 50;
        state[3] = -10;
    }
    public void init(double val[]) {
        state[0] = val[0]; // x
        state[1] = val[1]; // angle
        state[2] = val[2]; // y
        state[3] = val[3]; // direction
    }
    public double[] state() {
        return state;
    }
    public double[] compute(double val[]) {
        state[3] = val[1];
        state[1] += state[3]*val[0]/25;
        if( state[1] > 180) state[1] -= 360;
        if( state[1] < -180) state[1] += 360;
        state[0] += state[3]*Math.sin(state[1]*Math.PI/180)/10;
        state[2] += state[3]*Math.cos(state[1]*Math.PI/180)/10;
        return state;
    }
}

```

Once the plant model is described, the user must compile it to generate the .class binary file. Be aware of the value of the environment variable CLASSPATH, as it must contain the path to the interface definition. Hence, CLASSPATH must include the route "*base/xfuzzy.jar*", where base refers to the installation directory of Xfuzzy. (Note: in MS-Windows the path must include the route "*base\xfuzzy.jar*". Be aware of the separator).

The following figure shows a graphical representation corresponding to a simulation process on the *truck* system with the above commented plant model. A limit of 200 iterations has been imposed, and the initial state has been set to a (30,15) position and a 135-degree angle.



Tuning stage

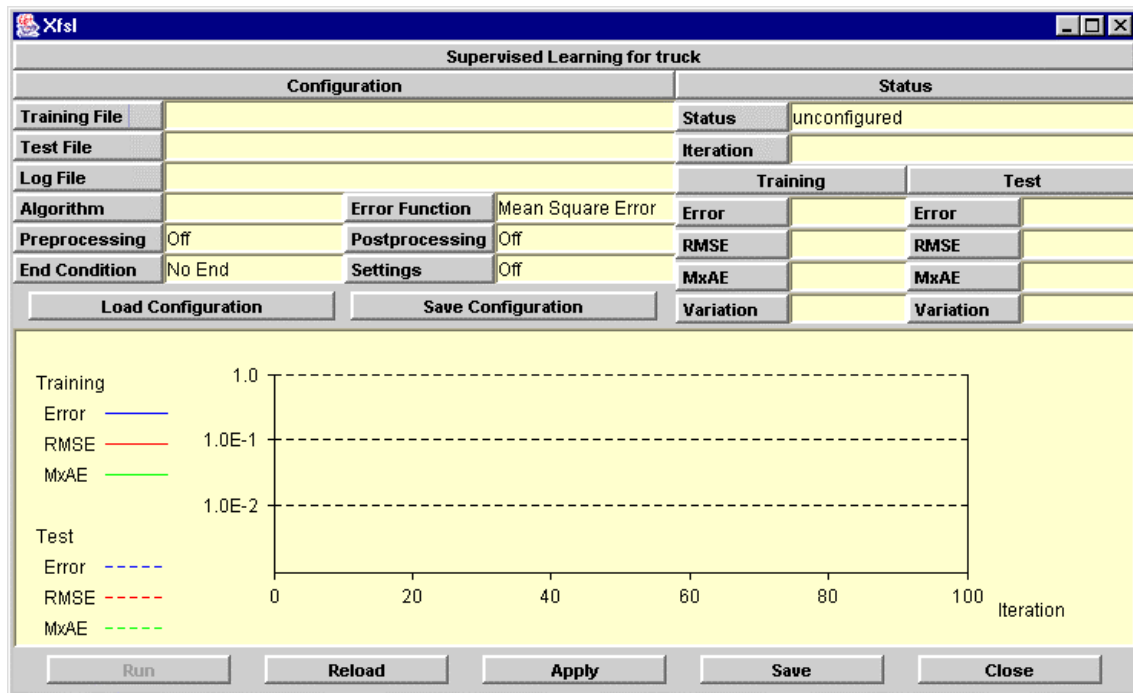
The tuning stage is usually one of the most complex tasks when designing fuzzy systems. The system behavior depends on the logic structure of its rule base and the membership functions of its linguistic variables. The tuning process is very often focused on adjusting the different membership function parameters that appear in the system definition. Since the number of parameters to simultaneously modify is high, a manual tuning is clearly cumbersome and automatic techniques are required. The two learning mechanisms most widely used are supervised and reinforcement learning. In supervised learning techniques the desired system behavior is given by a set of training (and test) input/output data while in reinforcement learning what is known is not the exact output data but the effect that the system has to produce on its environment, thus making necessary the monitoring of its on-line behavior.

Xfuzzy 3.0 environment currently contains one tool dedicated to the tuning stage: [xfsl](#), which is based on the use of supervised learning algorithms. Supervised learning attempts to minimize an error function that evaluates the difference between the actual system behavior and its desired behavior defined by a set of input/output patterns.

The supervised learning tool - Xfsl

Xfsl is a tool that allows the user to apply supervised learning algorithms to tune fuzzy systems into the design flow of Xfuzzy 3.0. The tool can be executed in graphical mode or in command mode. The graphical mode is used when executing the tool from the main window of the environment (using the option "*Supervised*

learning" in the *Tuning* menu). The command mode is used when executing the tool from the command line with the expression "xfs/ file.xfl file.cfg", where the first file contains the system definition in XFL3 format, and the second one contains the configuration of the learning process (see [configuration file](#) below).



The figure above illustrates the main window of *xfs/*. This window is divided into four parts. The left upper corner is the area to configure the learning process. The state of the learning process is shown at the right upper part. The central area illustrates the evolution of the learning, and the bottom part contains several control buttons to run or stop the process, to save the results, and to exit.

In order to configure the learning process, the first step is to select a training file that contains the input/output data of the desired behavior. A test file, whose data are used to check the generalization of the learning, can be also selected. The format of these two patterns files is just an enumeration of numeric values that are assigned to the input and output variables in the same order that they appear in the definition of the *system* module in the XFL3 description. This is an example of a pattern file for a fuzzy system with two inputs and one output:

```

0.00 0.00 0.5
0.00 0.05 0.622459
0.00 0.10 0.731059
...

```

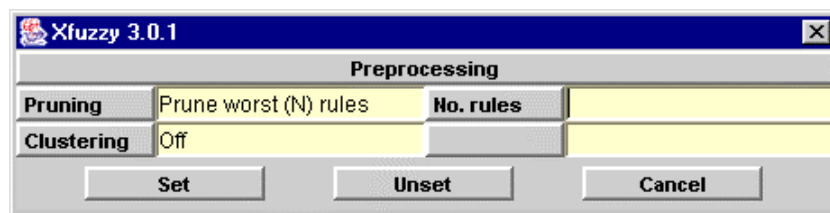
The log file allows to save the learning evolution in an external file. The selection of this file is optional.

The following step in the configuration of the tuning process is the selection of the learning algorithm. *Xfsl* admits many learning algorithms (see section [algorithms](#) below). Regarding gradient descent algorithms, it admits *Steepest Descent*, *Backpropagation*, *Backpropagation with Momentum*, *Adaptive Learning Rate*, *Adaptive Step Size*, *Manhattan*, *QuickProp* and *RProp*. Among conjugate gradient

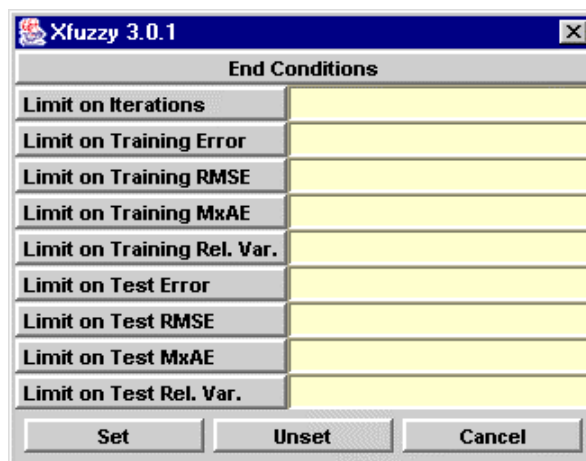
algorithms, the following are included: *Polak-Ribiere*, *Fletcher-Reeves*, *Hestenes-Stiefel*, *One-step Secant* and *Scaled Conjugate Gradient*. The second-order algorithms included are: *Broyden-Fletcher-Goldfarb-Shanno*, *Davidon-Fletcher-Powell*, *Gauss-Newton* and *Mardquardt-Levenberg*. Regarding algorithms without derivatives, the *Downhill Simplex* and *Powell's method* can be applied. Finally, the statistical algorithms included are *Blind Search* and *Simulated Annealing* (with linear, exponential, classic, fast, and adaptive annealing schemes).

Once the algorithm is selected, an error function must be chosen. The tool offers several error functions that can be used to express the deviation between the actual and the desired behavior (see section [error function](#) below). By default, the *Mean Square Error* is selected.

Xfsl contains two processing algorithms to simplify the designed fuzzy system. The first algorithm prunes the rules and reduces the membership functions that do not reach a significant activation or membership degree. There are three versions of the algorithm: pruning all rules that are never activated over a certain threshold, pruning the worst N rules, and pruning all rules except the best N ones. The second algorithm clusters the membership functions of the output variables. The number of clusters can be fixed to a certain quantity, or computed automatically. These two processing algorithms can be applied to the system before the tuning process (preprocessing option) or after it (postprocessing option).

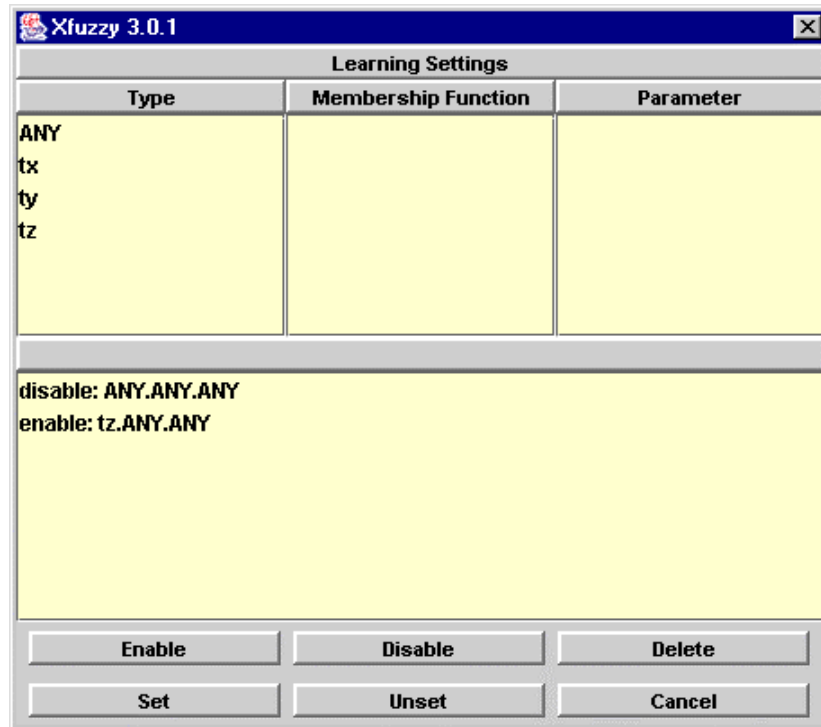


An end condition has to be specified to finish the learning process. This condition is a limit imposed over the number of iterations, the maximum error goal, or the maximum absolute or relative deviation (considering both the training or the test error).



The tool allows the user to choose the parameters to be tuned. The following window is used to enable or disable the tuning of the parameters. The three upper lists are used to select a parameter, or a set of parameters, by selecting the variable type, the membership function of that type, and the parameter index in that membership function. The lower list shows the actual settings. These settings are interpreted in the order that they appear in the list. In this example, all the

parameters are first disabled, and then the parameters of the type *tz* are enabled, so only the parameters of the *tz* type are going to be tuned.



A complete learning configuration can be saved into an external file that will be available for subsequent processes. The format of this file is described in section [configuration file](#).

Xfsl can be applied to any fuzzy system described by the XFL3 language, even to systems that employ particular functions defined by the user. What must be considered is that the features of the system may impose limitations over the learning algorithms to apply (for instance, a non derivative system can not be tuned by a gradient-descent algorithm).

Algorithms

Since the objective of supervised learning algorithms is to minimize an error function that summarizes the deviation between the actual and the desired system behavior, they can be considered as algorithms for function optimization. Xfsl contains many different supervised learning algorithms, which are briefly described in the following.

A) Gradient Descent Algorithms

The equivalence between fuzzy systems and neural networks led to apply the neural learning processes to fuzzy inference systems. In this sense, a well-known algorithm employed in fuzzy systems is the *BackPropagation* algorithm, which modifies the parameter values proportionally to the gradient of the error function in order to reach a local minimum. Since the convergence speed of this algorithm is slow, several modifications were proposed like using a different learning rate for each parameter or adapting heuristically the control variables of the algorithm. An interesting modification that improves greatly the convergence speed is to take into account the gradient value of two successive iterations because this provides

information about the curvature of the error function. The algorithms *QuickProp* and *RProp* follow this idea.

Xfsl admits *Backpropagation*, *Backpropagation with Momentum*, *Adaptive Learning Rate*, *Adaptive Step Size*, *Manhattan*, *QuickProp* and *RProp*.

B) Conjugate Gradient Algorithms

The gradient-descent algorithms generate a change step in the parameter values that is a function of the gradient value at each iteration (and possibly at previous iterations). Since the gradient indicates the direction of maximum function variation, it may be convenient to generate not only one step but several steps which minimize the function error in that direction. This idea, which is the basis of the steepest-descent algorithm, has the drawback of producing a zig-zag advancing because the optimization in one direction may deteriorate previous optimizations. The solution is to advance by conjugate directions that do not interfere each other. The several conjugate gradient algorithms reported in the literature differ in the equations used to generate the conjugate directions.

The main drawback of the conjugate gradient algorithms is the implementation of a linear search in each direction, which may be costly in terms of function evaluations. The linear search can be avoided by using second-order information, that is, by approximating the second derivative with two close first derivatives. The *scaled conjugate gradient* algorithm is based on this idea.

Among conjugate gradient algorithms, the following are included in *xfsl*: *Steepest Descent*, *Polak-Ribiere*, *Fletcher-Reeves*, *Hestenes-Stiefel*, *One-step Secant* and *Scaled Conjugate Gradient*.

C) Second-Order Algorithms

A forward step towards speeding up the convergence of learning algorithms is to make use of second-order information of the error function, that is, of its second derivatives or, in matricial form, of its Hessian. Since the calculus of the second derivatives is complex, one solution is to approximate the Hessian by means of the gradient values of successive iterations. This is the idea of *Broyden-Fletcher-Goldarfb-Shanno* and *Davidon-Fletcher-Powell* algorithms.

A particular case is when the function to minimize is a quadratic error because the Hessian can be approximated by only the first derivatives of the system outputs, as done by the *Gauss-Newton* algorithm. Since this algorithm can lead to instability when the approximated Hessian is not positive defined, the *Marquardt-Levenberg* algorithm solves this problem by introducing an adaptive term.

The second-order algorithms included in the tool are: *Broyden-Fletcher-Goldarfb-Shanno*, *Davidon-Fletcher-Powell*, *Gauss-Newton* and *Marquardt-Levenberg*.

D) Algorithms Without Derivatives

The gradient of the error function cannot be always calculated because it can be too costly or not defined. In these cases, optimization algorithms without derivatives can be employed. An example is the *Downhill Simplex* algorithm, which considers a set of function evaluations to decide a parameter change. Another example is *Powell's method*, which implements linear searches by a set of directions that evolve to be conjugate. The algorithms of this kind are too much slower than the

previous ones. A best solution can be to estimate the derivatives from the secants or to employ not the derivative value but its sign (as *RProp* does), which can be estimated from small perturbations of the parameters.

All the above commented algorithms do not reach the global but a local minimum of the error function. The statistical algorithms can discover the global minimum because they generate different system configurations that spread the search space. One way of broadening the space explored is to generate random configurations and choose the best one. This is done by the *Blind Search* algorithm, whose convergence speed is extremely slow. Another way is to perform small perturbations in the parameters to find a better configuration as done by the algorithm of iterative improvements. A better solution is to employ *Simulated Annealing* algorithms. They are based on an analogy between the learning process, which is intended to minimize the error function, and the evolution of a physical system, which tends to lower its energy as its temperature decreases. Simulated annealing provides good results when the number of parameters to adjust is low. When it is high, the convergence speed can be so slow that it can be preferred to generate random configurations, apply gradient descent algorithms and select the best solution.

Regarding algorithms without derivatives, the *Downhill Simplex* and *Powell's method* can be applied. The statistical algorithms included are *Blind Search* and *Simulated Annealing* (with linear, exponential, classic, fast, and adaptive annealing schemes).

When optimizing a differentiable system, *Broyden-Fletcher-Goldfarb-Shanno* and *Mardquardt-Levenberg* algorithms are the most adequate. When using BFGS, control values (0.1,10) may be a good choice. In ML algorithm, control values (0.1,10,0.1) are a good initial option. If it is not possible to compute the system derivatives, as in hierarchical fuzzy systems, the best choice is to use these algorithms with the option of estimating the derivative. *Simulated Annealing* is only recommended when there are a few parameters to tune and the second order algorithms drive the system to a non-optimal minimum.

Error function

The error function expresses the deviation between the actual behavior of the fuzzy system and the desired one by comparing the input/output patterns with the output of the system for those input values. *Xfsl* defines seven error functions:

mean_square_error (MSE), *weighted_mean_square_error* (WMSE), *mean_absolute_error* (MAE), *weighted_mean_absolute_error* (WMAE), *classification_error* (CE), *advanced_classification_error* (ACE), and *classification_square_error* (CSE).

All these function are normalized by the number of patterns, the number of output variables, and the range of each output variable, so that the range of the error function is from 0 to 1. The first four functions are adequate for systems with continuous output variables, while the last three functions are dedicated to classification systems. These are the equation for the first functions:

$$\begin{aligned} \text{MSE} &= \text{Sum} (((Y-y)/\text{range})^{**2}) / (\text{num_pattern} * \text{num_output}) \\ \text{WMSE} &= \text{Sum} (w * ((Y-y)/\text{range})^{**2}) / (\text{num_pattern} * \text{Sum}(w)) \\ \text{MAE} &= \text{Sum} (|((Y-y)/\text{range})|) / (\text{num_pattern} * \text{num_output}) \\ \text{WMAE} &= \text{Sum} (w * |((Y-y)/\text{range})|) / (\text{num_pattern} * \text{Sum}(w)) \end{aligned}$$

The output of a fuzzy classification system is the linguistic label that has the greatest activation degree. A common way of expressing the deviation of these systems is the number of classification failures (*classification_error*, CE). This is not a very good choice for tuning because many system configurations produce the same number of failures. A useful modification is to add a term that measures the distance of the selected label to the desired one (*advanced_classification_error*, ACE). These two error functions are not differentiable, so they cannot be used with derivative-based learning algorithms (which are the fastest). A better choice is to consider the activation degree of each linguistic label as the actual output and the desired output as 1 for the correct label and 0 for the others. The error function is computed as the square error of this system (*classification_square_error*, CSE), which is differentiable and can be used with derivative-based learning algorithms.

Configuration file

The configuration of a tuning process can be saved to and loaded from an external file. The contents of this file is formed by the following directives:

```
xfs1_training("file_name")
xfs1_test("file_name")
xfs1_log("file_name")
xfs1_output("file_name")
xfs1_algorithm(algorithm_name, value, value, ...)
xfs1_option(option_name, value, value, ...)
xfs1_errorfunction(function_name, value, value, ...)
xfs1_preprocessing(process_name, value, value, ...)
xfs1_postprocessing(process_name, value, value, ...)
xfs1_endcondition(condition_name, value, value, ...)
xfs1_enable(type.mf.number)
xfs1_disable(type.mf.number)
```

The directives *xfs1_training* and *xfs1_test* select the pattern files for training and testing the system. The log file for saving the learning evolution is selected by the directive *xfs1_log*. The directive *xfs1_output* contains the name of the XFL3 file to which the tuned system is saved. By default, this file is "*xfs1_out.xfl*".

The learning algorithm is set by the directive *xfs1_algorithm*. The values refer to the control variables of the algorithm. Once the algorithm has been chosen, any algorithm option can be selected by the directive *xfs1_option*.

The error function selection is made by the directive *xfs1_errorfunction*. The values contain the weights of the output variables for weighted error functions.

The directives *xfs1_preprocessing* and *xfs1_postprocessing* specify any process that has to be made before or after the system tuning. The different options are: *prune_threshold*, *prune_worst*, *prune_except*, and *output_clustering*. When option *output_clustering* contains a value, it refers to the number of clusters to be created, otherwise the number is computed automatically.

The end condition, selected by *xfs1_endcondition*, can be one of the following: *epoch*, *training_error*, *training_RMSE*, *training_MXAE*, *training_variation*, *test_error*, *test_RMSE*, *test_MXAE*, and *test_variation*.

The selection of the parameters to be tuned is made by the directives *xfs_enable* and *xfs_disable*. The fields *type*, *mf*, and *number* specify the variable type, membership function and index of the parameter. These fields can also contain the expression "ANY".

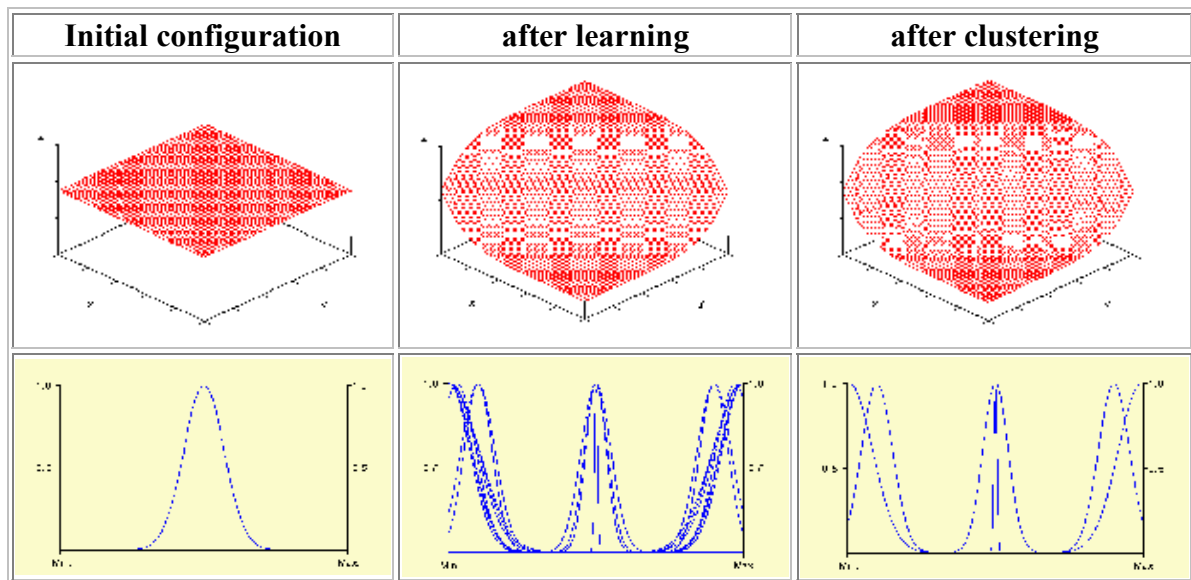
Example

The directory "examples/approxim/" contains some examples of tuning processes. The initial system configuration is specified in the file *plain.xfl*, which defines a fuzzy system with two input and one output variable. The system includes a type definition for each variable. The two input variable types contain seven bell membership functions, homogeneously distributed along the universe of discourse. The output variable type contains forty-nine bell membership functions that are equal, so the input/output behavior of this initial configuration is given by a flat surface.

The file *f1.trn* contains 441 patterns describing the surface:

$$z = 1 / (1 + \exp(10*(x-y)))$$

The following table shows the result of a tuning process using this training file. The learning algorithm used has been the *Marquardt-Levenberg* algorithm, with control values 0.1, 1.5 and 0.7. The tuning process has used the *Mean Square Error* and has been applied to all the system parameters. The tuning process changes mainly the output membership functions, making the system approximate the desired surface. As a result of the learning process, the output membership functions have formed several groups. Making a clustering postprocessing, the number of these functions can be reduced to six.



Synthesis stage

The synthesis stage is the last step in the design flow of a system. Its aim is to generate a system representation that could be used externally. There are two different types of final representations for a fuzzy system: software representations and hardware representations. The software synthesis generates a system representation in a high level programming language. The hardware synthesis generates a microelectronic circuit that implements the inference process described by the fuzzy system.

Software representations are useful when there are not strong restrictions on the inference speed, the system size, or the power consumption. They can be generated from any fuzzy system developed in Xfuzzy. On the other hand, hardware representations are more adequate when high speed, small area, or power is needed, but for this solution to be efficient some constraints has to be imposed on the fuzzy systems, so that the hardware synthesis is not so generic as its software counterpart.

Xfuzzy 3.0 provides the user with three tools for software synthesis: [xfc](#), that generates an ANSI-C description of the system, [xfcpp](#), to develop a C++ description, and [xfj](#), that represents the system as a Java class. The hardware synthesis capabilities of Xfuzzy are still under development.

The ANSI-C code generation tool - Xfc

The tool *xfc* generates an ANSI-C representation of the fuzzy system. The tool can be executed from the command line, with the expression "*xfc file.xfl*", or from the *Synthesis* menu in the main window of the environment. Since the generation of the ANSI-C representation does not need any additional information, this tool does not implement a graphical user interface.

Given the specification of a fuzzy system in the XFL3 format, *systemname.xfl*, the tool generates two files: *systemname.h*, containing the definition of the data structures, and *systemname.c*, containing the C functions that implement the fuzzy inference system. These files are generated into the same directory of the *systemname.xfl* file. The inference function can be used in external C projects by including the heading file (*systemname.h*) into them.

For a fuzzy system with global input variables *i0*, *i1*, ..., and global output variables *o0*, *o1*, ..., the inference function included in the *systemname.c* file is:

```
void systemnameInferenceEngine(double i0, double i1, ..., double *o0,
double *o1, ...);
```

The C++ code generation tool - Xfcpp

The tool *xfcpp* generates a C++ representation of the fuzzy system. The tool can be executed from the command line, with the expression "*xfcpp file.xfl*" or from the *Synthesis* menu in the main window of the environment. This tool neither has a graphical user interface because the generation of the C++ representation does not need any additional information.

Given the specification of a fuzzy system in the XFL3 format, *systemname.xfl*, the tool generates four files: *xfuzzy.hpp*, *xfuzzy.cpp*, *systemname.hpp*, and *systemname.cpp*. The files *xfuzzy.hpp* and *xfuzzy.cpp* contain the description of the C++ classes that are common to all fuzzy systems. The files *systemname.hpp* and *systemname.cpp* contain the description of the specific classes of the system. The files with '.hpp' extension are heading files that define the class structures, while the files with '.cpp' extension contain the body of the functions of each class. These files are generated into the same directory of the *systemname.xfl* file.

The C++ code generated by *xfcpp* develops a fuzzy inference engine that can be used with crisp values and fuzzy values. A fuzzy value is encapsulated into a *MembershipFunction* class object.

```
class MembershipFunction {
public:
    enum Type { GENERAL, CRISP, INNER };
    virtual enum Type getType() { return GENERAL; }
    virtual double getValue() { return 0; }
    virtual double compute(double x) = 0;
    virtual ~MembershipFunction() {}
};
```

The class describing the fuzzy system is an extension of the abstract class *FuzzyInferenceEngine*. This class, defined in *xfuzzy.hpp*, contains four methods that implement the fuzzy inference process.

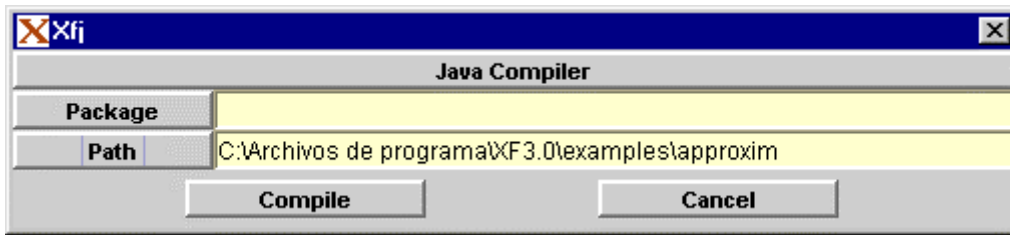
```
class FuzzyInferenceEngine {
public:
    virtual double* crispInference(double* input) = 0;
    virtual double* crispInference(MembershipFunction* &input) = 0;
    virtual MembershipFunction** fuzzyInference(double* input) = 0;
    virtual MembershipFunction** fuzzyInference(MembershipFunction*
&input) = 0;
};
```

The file *systemname.cpp* contains the description of the *systemname* class, which implements the fuzzy inference process for the system. Besides describing the four methods of the *FuzzyInferenceEngine* class, the system class contains a method, called *inference*, which develops the inference process with variables instead of arrays of variables. For a fuzzy system with global input variables *i0*, *i1*, ..., and global output variables *o0*, *o1*, ..., the inference function is:

```
void inference(double i0, double i1, ..., double *o0, double *o1, ...);
```

The Java code generation tool - Xfj

The tool *xfj* generates a Java representation of the fuzzy system. The tool can be executed from the command line, with the expression "*xfj [-p package] file.xfl*" or from the *Synthesis* menu in the main window of the environment. When invoked from the command line no graphical interface is shown. In this case, the Java code files are generated in the directory of the system file, and a *package* instruction is added in the Java classes when option *-p* is used. When the tool is invoked from the Xfuzzy main window, the package name and the target directory can be chosen in the following window.



Given the specification of a fuzzy system in the XFL3 format, *systemname.xfl*, the tool generates four files: *FuzzyInferenceEngine.java*, *MembershipFunction.java*, *FuzzySingleton.java*, and *systemname.java*. The first three files are descriptions of two interfaces and one class that are common to all fuzzy inference systems, while the last one contains the specific description of the fuzzy system.

The file *FuzzyInferenceEngine.java* describes a Java interface defining a general fuzzy inference system. This interface defines four methods to implement the inference process with crisp and fuzzy values.

```
public interface FuzzyInferenceEngine {
    public double[] crispInference(double[] input);
    public double[] crispInference(MembershipFunction[] input);
    public MembershipFunction[] fuzzyInference(double[] input);
    public MembershipFunction[] fuzzyInference(MembershipFunction[]
input);
}
```

The file *MembershipFunction.java* contains the description of an interface used to describe a fuzzy number. It has just one method, called *compute*, which computes the membership degree for each value of the universe of discourse of the fuzzy number.

```
public interface MembershipFunction {
    public double compute(double x);
}
```

The class *FuzzySingleton* implements the *MembershipFunction* interface, and represents a crisp value as a fuzzy number.

```
public class FuzzySingleton implements MembershipFunction {
    private double value;

    public FuzzySingleton(double value) { this.value = value; }
    public double getValue() { return this.value; }
    public double compute(double x) { return (x==value? 1.0: 0.0); }
}
```

Finally, the *systemname.java* contains the class which describes the fuzzy system. This class is an implementation of the interface *FuzzyInferenceEngine*. Hence, the public methods which develops the inference are those of the interface (*crispInference* and *fuzzyInference*).