



# HERRAMIENTAS DE CAD PARA LÓGICA DIFUSA

**X fuzzy 25<sup>th</sup>**

[xfuzzy-team@imse-cnm.csic.es](mailto:xfuzzy-team@imse-cnm.csic.es)

©IMSE-CNM 2018

Copyright (c) 2018, Instituto de Microelectrónica de Sevilla (IMSE-CNM)

Todos los derechos reservados.

Se permite la redistribución y el uso en formatos fuente y binario, con o sin modificaciones, siempre que se cumplan las siguientes condiciones:

- Las redistribuciones del código fuente deben incluir la anterior nota de copyright, esta lista de condiciones y la siguiente renuncia de responsabilidad.
- Las redistribuciones en forma binaria deben reproducir la anterior nota de copyright, esta lista de condiciones y la siguiente renuncia de responsabilidad en la documentación y/o otros materiales provistos con la distribución.
- Ni el nombre del IMSE-CNM ni los nombres de sus colaboradores pueden ser usados para aprobar o promover productos derivados de este software sin un permiso escrito específico previo.

ESTE SOFTWARE ES PROPORCIONADO POR LOS TITULARES DE DERECHOS DE AUTOR Y SUS COLABORADORES "TAL CUAL ES" Y TODAS LAS GARANTÍAS EXPRESAS O IMPLÍCITAS, INCLUYENDO, PERO NO LIMITÁNDOSE A, LAS GARANTÍAS IMPLÍCITAS DE COMERCIALIZACIÓN E IDONEIDAD PARA UN PROPÓSITO PARTICULAR SON DENEGADAS. EN NINGÚN CASO LOS TITULARES DE DERECHOS DE AUTOR O SUS COLABORADORES PODRÁN SER RESPONSABLES POR CUALQUIER DAÑO DIRECTO, INDIRECTO, INCIDENTAL, ESPECIAL, EJEMPLAR, O RESULTANTES (INCLUYENDO, PERO NO LIMITÁNDOSE A, PROCURACIÓN DE BIENES O SERVICIOS SUSTITUTOS; PÉRDIDA DE FUNCIONALIDAD, DATOS, O BENEFICIOS; O INTERRUPCIÓN DE NEGOCIOS) SIN IMPORTAR SU CAUSA Y BAJO ALGUNA TEORÍA DE RESPONSABILIDAD, YA SEA POR CONTRATO, RESPONSABILIDAD LIMITADA, O AGRAVIOS (INCLUYENDO NEGLIGENCIA O CUALQUIER OTRO) RESULTANTE DE CUALQUIER MODO DE USO DE ESTE SOFTWARE, INCLUSO SI SE LE ADVIRTIÓ DE LA POSIBILIDAD DE DICHO DAÑO.

## Tabla de Contenidos

• Notas de la versión 3.5 .....	<a href="#">4</a>
• Instalación de <i>Xfuzzy</i> 3.5 .....	<a href="#">7</a>
• Introducción a <i>Xfuzzy</i> 3 .....	<a href="#">8</a>
• XFL3: El lenguaje de especificación de <i>Xfuzzy</i> 3 .....	<a href="#">9</a>
○ Conjunto de operadores .....	<a href="#">10</a>
○ Tipos de variables lingüísticas .....	<a href="#">11</a>
○ Bases de reglas .....	<a href="#">12</a>
○ Bloques no difusos .....	<a href="#">14</a>
○ Comportamiento global del sistema .....	<a href="#">15</a>
○ Paquetes de funciones	
▪ Definición de <a href="#">funciones binarias</a>	
▪ Definición de <a href="#">funciones unarias</a>	
▪ Definición de <a href="#">funciones no difusas</a>	
▪ Definición de <a href="#">funciones de pertenencia</a>	
▪ Definición de <a href="#">familias de funciones de pertenencia</a>	
▪ Definición de <a href="#">métodos de defuzzificación</a>	
▪ El <a href="#">paquete estándar xfl</a>	
• Entorno de desarrollo <i>Xfuzzy</i> 3 .....	<a href="#">37</a>
○ Etapa de descripción .....	<a href="#">38</a>
▪ Edición de sistemas ( <a href="#">xfedit</a> )	
▪ Edición de paquetes ( <a href="#">xfpkg</a> )	
○ Etapa de verificación .....	<a href="#">52</a>
▪ Representación gráfica ( <a href="#">xfplot</a> )	
▪ Monitor de inferencias ( <a href="#">xfmt</a> )	
▪ Simulación de sistemas ( <a href="#">xfsim</a> )	
○ Etapa de ajuste .....	<a href="#">62</a>
▪ Adquisición de conocimiento ( <a href="#">xfdm</a> )	
▪ Predicción de series temporales ( <a href="#">xftsp</a> )	
▪ Aprendizaje supervisado ( <a href="#">xfsl</a> )	
▪ Simplificación ( <a href="#">xfsp</a> )	
○ Etapa de síntesis .....	<a href="#">80</a>
▪ Generación de código C ( <a href="#">xfc</a> )	
▪ Generación de código C++ ( <a href="#">xfcpp</a> )	
▪ Generación de código Java ( <a href="#">xfj</a> )	
▪ Generación de código VHDL ( <a href="#">xfvhdl</a> )	
▪ Generación de modelos SysGen ( <a href="#">xfsg</a> )	

## Notas de la versión 3.5

### Cambios en la versión 3.5 con respecto a la 3.3

- Nueva Funcionalidad:
  1. La interfaz gráfica de **Xfuzzy** muestra ahora las especificaciones mediante estructuras desplegadas, de forma que es posible seleccionar el sistema completo o cualquiera de sus bases de reglas como la especificación activa sobre la que actuarán las distintas herramientas.
  2. Se ha integrado en el entorno la herramienta de predicción de series temporales, **xftsp**, a la que puede accederse a través del menú *Tuning* de la ventana principal de *Xfuzzy*.
  3. Se ha añadido en el menú *File* de **xfplot** la opción *Save image*, que permite guardar la representación grafica en un fichero JPEG.
  4. La herramienta de síntesis hardware **xfvhdl** ha sido actualizada para que genere ficheros de síntesis para los entornos de diseño de FPGAs ISE y Vivado de Xilinx.
  5. Todas las herramientas del entorno *Xfuzzy* pueden ser invocadas desde la línea de comandos.
- Documentación y material didáctico:
  1. Se ha actualizado y completado la documentación del entorno **Xfuzzy**, de forma que describa la funcionalidad de todas las herramientas que lo integran.
  2. Se han incluido, como parte de la distribución de **Xfuzzy**, ejemplos de uso de las distintas facilidades del entorno de forma independiente (*Tools*), así como en combinación con otras herramientas informáticas para desarrollar diferentes aplicaciones (*Apps*).
  3. En la página web de **Xfuzzy** se encuentran disponibles asimismo una serie de tutoriales que detallan el uso de las herramientas de síntesis hardware del entorno para aplicar distintas metodologías de desarrollo de controladores difusos sobre FPGAs de Xilinx.
- Problemas corregidos:
  1. Se ha unificado el idioma de las ventanas del sistema utilizadas para localizar archivos y directorios, de forma que todas las leyendas aparezcan en inglés.
  2. Se ha corregido un fallo que impedía editar paquetes de funciones con la herramienta **xfpkg**.
  3. Se han depurado varios errores en la ejecución de determinados algoritmos de identificación utilizados por la herramienta **xfdm**.
  4. Se han eliminado las directivas de configuración de la herramienta **xftsp** que no tenían uso.

5. Se ha corregido un error que presentaba la herramienta **xfsim** al cargar el modelo de la planta por problemas con el camino de búsqueda del fichero.
6. Se ha modificado el código c++ generado por la herramienta **xfc++** para hacerlo compatible con los compiladores gcc disponibles en distintas distribuciones de Linux y con el compilador de Visual Studio para Windows.

## Cambios en la versión 3.3 con respecto a la 3.0

- Se han incluido en el entorno dos nuevas herramientas de síntesis hardware:
  1. Xfvhd traslada la especificación de un sistema difuso escrita en XFL3 en una descripción VHDL que puede ser sintetizada e implementada sobre un dispositivo programable o como un circuito integrado para aplicaciones específicas.

Comparada con las versiones previas de las herramientas de síntesis hardware incluidas en *Xfuzzy*, las principales novedades de la nueva versión de xfvhdl son:

- Permite la implementación directa de sistemas difusos jerárquicos.
  - Se ha mejorado la funcionalidad de muchos de los componentes de la librería XHDL incluida en esta nueva versión. Los circuitos aritméticos han sido modificados para que generen las regiones de saturación para funciones de pertenencia de tipo "Z" y "S". Se ha introducido un nuevo bloque que implementa el método de defuzzificación Takagi-Sugeno de primer orden. La librería incluye también nuevos bloques crisp que implementan funciones aritméticas de propósito general (suma, resta, multiplicación y división) y operaciones lógicas (selector).
  - Las descripciones VHDL de la librería de componentes han sido parametrizadas mediante sentencias VHDL de tipo "generic" con objeto de facilitar la automatización del proceso de diseño.
  - Se ha desarrollado una interfaz gráfica mejorada para incluir la nueva funcionalidad de la herramienta.
2. Xfsg traslada la especificación XFL de un sistema difuso en un modelo Simulink que incluye componentes de la librería XfuzzyLib. En combinación con las herramientas de implementación de FPGAs de Xilinx y las facilidades de simulación de Matlab, esta herramienta proporciona un potente entorno para la síntesis de sistemas de inferencia difusos sobre FPGAs de Xilinx.

## Cambios en la versión 3.0 con respecto a la 2.X

1. El entorno ha sido completamente reprogramado usando Java.
2. Se ha definido un nuevo lenguaje de especificación de sistemas difusos, XFL3. Algunas de las mejoras con respecto a XFL son las siguientes:
  1. Se ha incorporado una nueva clase de objeto, llamado "*operator set*", para asignar funciones diferentes a los operadores difusos.

2. Se han incluido también modificadores lingüísticos (*Linguistic hedges*) que permiten describir relaciones más complejas entre variables lingüísticas.
3. El usuario puede ahora extender no sólo las funciones asignadas a los conectivos difusos y a los métodos de defuzzificación sino también las funciones de pertenencia y los modificadores lingüísticos.
3. La herramienta de edición permite ahora definir bases de reglas jerárquicas.
4. Las herramientas de representación en 2-D y 3-D no requieren el uso de gnuplot.
5. Se ha incorporado una nueva herramienta de monitorización para estudiar el comportamiento del sistema.
6. La herramienta de ajuste incluye muchos nuevos algoritmos de aprendizaje.

### **Problemas detectados en la versión 3.0**

1. (xfedit) La edición de funciones de pertenencia provoca a veces el error "Label already exists".
2. (xfedit) La edición de bases de reglas da error al aplicar las modificaciones dos veces.
3. (xfedit, xfmt) La estructura jerárquica del sistema no se dibuja correctamente cuando una variable interna se utiliza como entrada de una base de reglas y como variable de salida.
4. (xfsim) Las condiciones de fin sobre las variables de entrada del sistema no se verifican correctamente.
5. (tools) La ejecución en modo comando de las distintas herramientas no admite caminos absolutos para identificar los ficheros.
6. (XFL3) La utilización de un método de defuzzificación no verifica la cláusula "definedfor".
7. (xfcpp) Algunos compiladores no admiten que los métodos de la clase Operatorset se denominen "and", "or" o "not".
8. (xfsl) El proceso de clustering a veces genera nuevas funciones de pertenencia cuyos parámetros no cumplen las restricciones por errores de redondeo.
9. (tools) En ocasiones algunas ventanas de las herramientas no se dibujan correctamente y es necesario modificar el tamaño de estas ventanas para forzar una representación correcta.

## Instalación de Xfuzzy 3.5

### Requisitos del sistema:

*Xfuzzy* 3.5 puede ser ejecutado sobre cualquier plataforma que disponga del "Java Runtime Environment" (JRE). Para definir nuevos paquetes de funciones es también necesario disponer de un compilador Java. La última versión del "Java Software Development Kit", incluyendo el JRE, un compilador Java y otras herramientas relacionadas, puede encontrarse en <http://www.oracle.com/technetwork/java/>.

### Guía de Instalación:

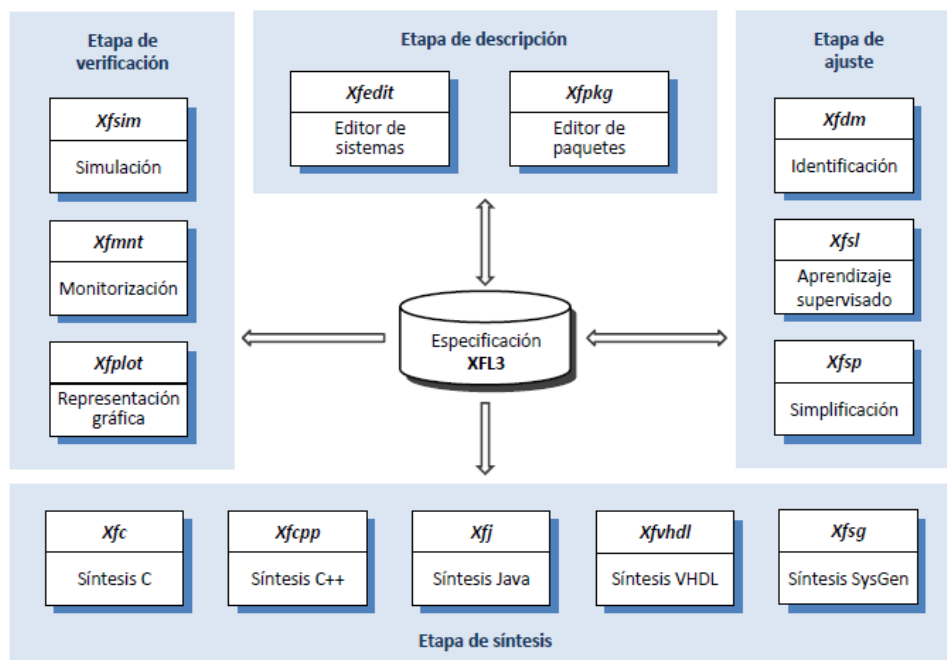
- Descargar el fichero [XfuzzyInstall.jar](#).
- Ejecutar este fichero. Sobre MS-Windows basta con pulsar sobre el icono. En general el fichero se ejecuta con el comando "`java -jar XfuzzyInstall.jar`". Esto abrirá la siguiente ventana de instalación.



- Elegir un directorio para instalar *Xfuzzy*. Si el directorio no existe se creará en el proceso de instalación.
- Elegir el directorio de los ejecutables de java (java, javac, jar, etc.). Este directorio suele ser el subdirectorio `"/bin"` de la instalación de Java.
- Elegir un navegador para mostrar los ficheros de ayuda.
- Pulsar en el botón "Install" para descomprimir la distribución de *Xfuzzy* en el directorio base seleccionado.
- Los ejecutables de *Xfuzzy* residen en el directorio `"/bin"`.
- Los ficheros ejecutables son ficheros de comandos. Si se cambia la localización de la distribución de *Xfuzzy* deberá repetirse el proceso de instalación.

## Introducción a Xfuzzy 3

*Xfuzzy 3* es un entorno de desarrollo para sistemas de inferencia basados en lógica difusa. Está formado por varias herramientas que cubren las diferentes etapas del proceso de diseño de sistemas difusos, desde su descripción inicial hasta la implementación final. Sus principales características son la capacidad para desarrollar sistemas complejos y la flexibilidad para permitir al usuario extender el conjunto de funciones disponibles. El entorno ha sido completamente programado en Java, de forma que puede ser ejecutado sobre cualquier plataforma que tenga instalado el JRE (*Java Runtime Environment*). La siguiente figura muestra el flujo de diseño de *Xfuzzy 3*.



La [etapa de descripción](#) incluye herramientas gráficas para la definición del sistema difuso. La [etapa de verificación](#) está compuesta por herramientas de simulación, monitorización y representación gráfica del comportamiento del sistema. La [etapa de ajuste](#) facilita la aplicación de algoritmos de aprendizaje. Finalmente, la [etapa de síntesis](#) incluye herramientas para generar descripciones en lenguajes de alto nivel para implementaciones software o hardware.

El nexo entre todas las herramientas es el uso de un lenguaje de especificación común, [XFL3](#), que extiende las capacidades de XFL, el lenguaje definido en la versión 2.0 de *Xfuzzy*. XFL3 es un lenguaje flexible y potente, que permite expresar relaciones muy complejas entre variables difusas por medio de bases de reglas jerárquicas, conectivos y modificadores lingüísticos, funciones de pertenencia y métodos de defuzzificación definidos por el usuario.

Las diferentes herramientas pueden ser ejecutadas como programas independientes. El entorno integra a todas ellas bajo una [interfaz gráfica de usuario](#) que facilita el proceso de diseño.



## XFL3: El lenguaje de especificación de Xfuzzy 3

- XFL3: El lenguaje de especificación de Xfuzzy 3
  - [Conjunto de operadores](#)
  - [Tipos de variables lingüísticas](#)
  - [Bases de reglas](#)
  - [Bloques no difusos](#)
  - [Comportamiento global del sistema](#)
  - [Paquetes de funciones](#)
    - Definición de funciones [binarias](#)
    - Definición de funciones [unarias](#)
    - Definición de funciones [no difusas](#)
    - Definición de [funciones de pertenencia](#)
    - Definición de [familias](#) de funciones de pertenencia
    - Definición de métodos de [defuzzificación](#)
    - El paquete estándar [xfl](#)

La definición de lenguajes formales para la especificación de sistema difusos presenta varias ventajas. Sin embargo, pueden plantearse dos objetivos contradictorios. Por una parte es deseable disponer de un lenguaje genérico y altamente expresivo, capaz de aplicar todos los formalismos basados en lógica difusa, pero, al mismo tiempo, las (posibles) restricciones impuestas por la implementación final del sistema deben ser consideradas. En este sentido, algunos lenguajes están enfocados hacia la expresividad, mientras otros están enfocados hacia las implementaciones software o hardware.

Uno de nuestros principales objetivos al comenzar a desarrollar un entorno de diseño de sistemas difusos fue obtener un entorno abierto, que no estuviera restringido por los detalles de implementación, pero que ofreciera al usuario un amplio conjunto de herramientas que permitieran diferentes implementaciones a partir de una descripción general del sistema. Esto nos llevó a la definición del lenguaje formal XFL. Las principales características de XFL fueron la separación entre la definición de la estructura del sistema y la definición de las funciones asignadas a los operadores difusos, y su capacidad para definir sistemas complejos. XFL es la base de varias herramientas de desarrollo orientadas al hardware y al software que constituyen el entorno de diseño *Xfuzzy 2.0*.

Como punto de partida para la versión 3 de *Xfuzzy*, ha sido definido un nuevo lenguaje, XFL3, que extiende las ventajas de XFL. XFL3 permite al usuario definir nuevas funciones de pertenencia y operadores paramétricos, y admite el uso de modificadores lingüísticos que permiten describir relaciones más complejas entre las variables. Con objeto de incorporar estas mejoras se han introducido algunas modificaciones en la sintaxis de XFL. Además, el nuevo lenguaje XFL3, así como las herramientas basadas en él, emplean Java como lenguaje de programación. Esto significa el uso de una ventajosa metodología orientada a objetos y flexibilidad para ejecutar la nueva versión de *Xfuzzy* en cualquier plataforma que tenga instalado JRE (*Java Runtime Environment*).

XFL3 divide la descripción de un sistema difuso en dos partes: la definición lógica de la estructura del sistema, que es incluida en ficheros con extensión ".xfl", y la definición matemática de las funciones difusas, que son incluidas en ficheros con extensión ".pkg" (*packages*).

El lenguaje permite la definición de sistemas complejos. XFL3 no limita el número de variables lingüísticas, funciones de pertenencia, reglas difusas, etc. Los sistemas pueden ser definidos mediante una jerarquía de módulos (bases de reglas o bloques no difusos) y las bases de reglas pueden expresar relaciones complejas entre las variables lingüísticas usando los conectivos AND y OR y modificadores lingüísticos como mayor que, más pequeño que, distinto a, etc. XFL3 permite al usuario definir sus propias funciones difusas por medio de paquetes ([packages](#)). Estas nuevas funciones pueden ser usadas como funciones de pertenencia, familias de funciones de pertenencia, conectivos difusos, modificadores lingüísticos, bloques no difusos y métodos de defuzzificación. El paquete estándar [xfl](#) contiene las funciones más habituales.

La descripción de la estructura de un sistema difuso, incluida en ficheros ".xfl", emplea una sintaxis formal basada en 8 palabras reservadas: *operatorset*, *type*, *extends*, *rulebase*, *using*, *if*, *crisp* y *system*. Una especificación XFL3 contiene varios objetos que definen conjuntos de operadores, tipos de variables, bases de reglas, bloques no difusos y la descripción del comportamiento global del sistema. Un conjunto de operadores ([operator set](#)) describe la selección de las funciones asignadas a los diferentes operadores difusos. Un [tipo de variable](#) contiene la definición del universo de discurso, las etiquetas lingüísticas y las funciones de pertenencia relacionadas con una variable lingüística. Una [base de reglas](#) define las relaciones lógicas entre las variables lingüísticas. Un [bloque no difuso](#) describe una operación matemática entre variables del sistema. Por último, el [comportamiento global del sistema](#) incluye la descripción de la jerarquía de bases de reglas.

## Conjuntos de operadores

Un conjunto de operadores (*operator set*) en XFL3 es un objeto que contiene las funciones matemáticas asignadas a cada operador difuso. Los operadores difusos pueden ser binarios (como las T-normas y S-normas empleadas para representar conectivos entre variables lingüísticas, implicaciones o agregaciones de reglas), unarios (como las C-normas y los operadores relacionados con los modificadores lingüísticos), o pueden estar asociados con métodos de defuzzificación.

XFL3 define los conjuntos de operadores mediante el siguiente formato:

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

No es necesario especificar todos los operadores. Cuando uno de ellos no está definido, su valor por defecto es asumido. La siguiente tabla muestra los operadores (y sus funciones por defecto) actualmente usados en XFL3.

Operador	Tipo	Función por defecto
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

Las funciones asignadas son definidas en ficheros externos a los que llamamos paquetes (*packages*). El formato para identificar una función es "*package.function*".

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

## Tipos de variables lingüísticas

Un tipo XFL3 es un objeto que describe un tipo de variable lingüística. Esto significa definir su universo de discurso, dar nombre a las etiquetas lingüísticas que cubren dicho universo y especificar las funciones de pertenencia asociadas a cada etiqueta. El formato de definición de un tipo es el siguiente:

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

donde *min* y *max* son los límites del universo de discurso y *card* (cardinalidad) es su número de elementos discretos. Si la cardinalidad no es especificada se asume su valor por defecto (actualmente, 256). Cuando no se definen explícitamente los límites, el universo de discurso es considerado entre 0 a 1.

Las etiquetas lingüísticas pueden definirse de dos maneras: como funciones de pertenencia libres o como miembros de una familia de funciones de pertenencia. En este último caso, la familia de funciones de pertenencia debe haber sido definida previamente. Una función de pertenencia libre utiliza su propio conjunto de parámetros mientras que los miembros de las familias de funciones de pertenencia comparten la lista de parámetros de la familia. Esto resulta muy útil tanto para reducir el número de parámetros como para representar restricciones entre las etiquetas lingüísticas (como el orden o un grado de solapamiento fijo).

El formato del identificador de las funciones de pertenencia libres (*membership\_function*) y de las familias de funciones de pertenencia (*membership\_function\_family*) es similar al del identificador de un operador, es decir, "*package.function*". Por otro lado, las etiquetas

lingüísticas definidas como miembros de una familia se identifican por su índice dentro de la familia (comenzando por 0).

XFL3 soporta mecanismos de herencia en las definiciones de tipos (como su precursor XFL). La cabecera de la definición utilizada para expresar herencia es como sigue:

```
type identifier extends identifier {
```

Los tipos definidos de esta manera heredan automáticamente el universo de discurso y las etiquetas de sus padres. Las etiquetas definidas en el cuerpo de la definición del tipo son añadidas a las etiquetas de sus padres o sobrescriben a éstas si tienen los mismos nombres.

<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>	
<pre>type Tinput3 [-90,90] {   fam[] xfl.triangular(-60,- 30,0,30,60);   NG fam[0];   NM fam[1];   NP fam[2];   CE fam[3];   PP fam[4];   PM fam[5];   PG fam[6]; }</pre>	

## Bases de reglas

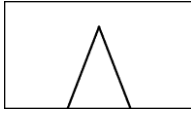
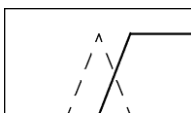

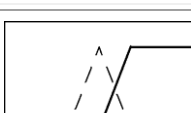

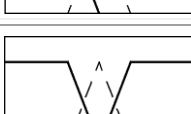
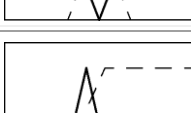
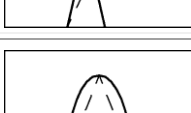
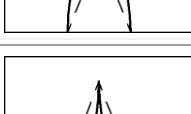
Una base de reglas en XFL3 es un objeto que contiene las reglas que definen las relaciones lógicas entre las variables lingüísticas. Su formato de definición es el siguiente:

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

El formato de definición de las variables de entrada y salida es "*type identifier*", donde *type* hace referencia a uno de los tipos de variables lingüísticas previamente definidas. La selección del conjunto de operadores es opcional, de forma que cuando no es definido explícitamente se emplean los operadores por defecto. Es posible aplicar a las reglas pesos o factores de confianza (con valor por defecto de 1).

El antecedente de una regla describe la relación entre las variables de entrada. XFL3 permite expresar antecedentes complejos combinando proposiciones básicas mediante conectivos y modificadores lingüísticos. Por otra parte, cada consecuente de una regla describe la asignación de un valor lingüístico a una variable de salida como "*variable = label*".

Una proposición básica relaciona una variable de entrada con una de sus etiquetas lingüísticas. XFL3 admite diferentes relaciones como igualdad, desigualdad y varios modificadores lingüísticos. La siguiente tabla muestra las diferentes relaciones ofrecidas por XFL3.

Proposiciones básicas	Descripción	Representación
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	

En general, el antecedente de una regla está formado por una proposición compleja. Las proposiciones complejas están compuestas de varias proposiciones básicas conectadas mediante conectivos difusos y modificadores lingüísticos. La siguiente tabla muestra cómo generar proposiciones complejas en XFL3.

Proposiciones complejas	Descripción
proposition & proposition	and operator
proposition   proposition	or operator
!proposition	not operator
%proposition	slightly operator
~proposition	moreorless operator
+proposition	strongly operator

Éste es un ejemplo de base de reglas compuesta por algunas reglas que incluyen proposiciones complejas.

```
rulebase base1(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;
  ..... }
```

## Bloques no difusos

Un bloque no difuso es un módulo que describe una operación no difusa entre algunas variables. En general, estos bloques suelen consistir en operaciones sencillas como la suma, la diferencia o el producto. Este tipo de operaciones matemáticas suelen encontrarse en problemas reales cuando es necesario adaptar las variables del sistema para que puedan ser utilizadas por alguna base de reglas o para generar una salida.

La definición de los bloques no difusos se encapsula en un objeto XFL llamado *crisp*. Una especificación de un sistema en XFL3 sólo puede contener la descripción de un objeto *crisp*. El formato de definición del objeto *crisp* es el siguiente:

```
crisp {
  identifier crisp_function(parameter_list);
  identifier crisp_function(parameter_list);
  ..... }
```

El formato del identificador del bloque no difuso (*crisp\_function*) es similar al del identificador de un operador, es decir, "*package.function*" o simplemente "*function*" si el paquete donde el usuario ha definido las funciones de pertenencia ha sido importado previamente:

```
crisp {
  difference xfl.diff2();
  summation xfl.addN(3);
}
```

## Comportamiento global del sistema

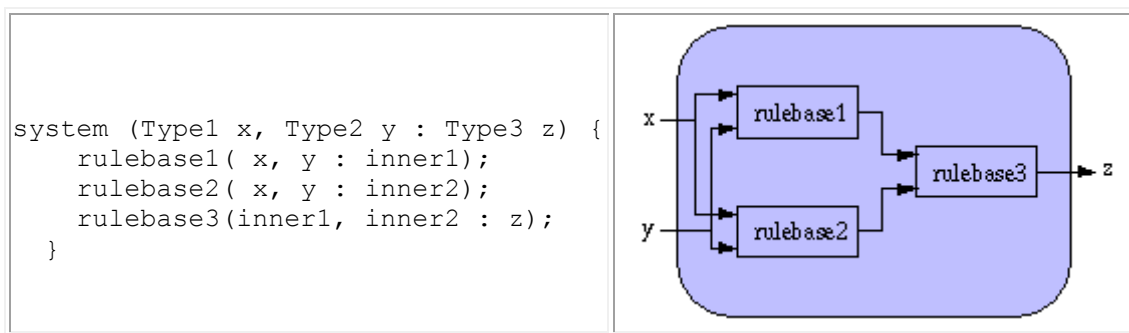
La descripción del comportamiento global del sistema requiere definir las variables globales de entrada y salida del sistema, así como la jerarquía modular. Esta descripción en XFL3 es como sigue:

```

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

```

El formato de definición de las variables de entrada y salida globales es el mismo que el empleado en la definición de las bases de reglas. Las variables internas que pueden aparecer establecen interconexiones en serie o en paralelo entre los módulos. Las variables internas deben aparecer como variables de salida de un módulo antes de ser empleadas como variables de entrada de otros módulos. Los módulos se refieren tanto a bases de reglas como a bloques no difusos.



## Paquetes de funciones

Una de las grandes ventajas de XFL3 es que las funciones asignadas a los operadores difusos pueden ser definidas libremente por el usuario en ficheros externos (denominados paquetes o "*packages*"), lo que proporciona una enorme flexibilidad al entorno. Cada package puede incluir un número ilimitado de definiciones.

En XFL3 pueden definirse seis tipos de funciones: [funciones binarias](#) que pueden ser usadas como T-normas, S-normas y funciones de implicación; [funciones unarias](#) que están relacionadas con los modificadores lingüísticos; [funciones no difusas](#) que desarrollan los bloques no difusos; [funciones de pertenencia](#) que son usadas para describir etiquetas lingüísticas; [familias de funciones de pertenencia](#) que describen conjuntos de funciones de pertenencia que comparten los parámetros; y [métodos de defuzzificación](#).

Una definición de función incluye su nombre (y posibles alias), los parámetros que definen su comportamiento junto con las restricciones de estos parámetros, la descripción de su comportamiento en los diferentes lenguajes en los que puede ser compilado (C, C++ y Java) e, incluso, la descripción de las derivadas de la función (si va a ser utilizada con mecanismos de aprendizaje basados en gradiente). Esta información es la base para generar automáticamente una clase Java que incorpora todas las capacidades de la función y puede ser empleada por cualquier especificación XF3.

## Definición de funciones binarias

Las funciones binarias pueden ser asignadas al operador de conjunción (and), al operador de disyunción (or), a la función de implicación (imp) y al operador de agregación de reglas (also). La estructura de una definición de función binaria en un paquete de funciones es como sigue:

```
binary identifier { blocks }
```

Los bloques que pueden aparecer en la definición de una función binaria son *alias*, *parameter*, *requires*, *java*, *ansi\_c*, *cplusplus*, *derivative* y *source*.

El bloque *alias* se utiliza para definir nombres alternativos para identificar a la función. Cualquiera de esos identificadores puede ser usado para hacer referencia a la función. La sintaxis del bloque *alias* es:

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

El bloque *parameter* permite la definición de los parámetros de los que depende la función. Su formato es:

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

El bloque *requires* expresa las restricciones sobre los valores de los parámetros por medio de una expresión Booleana en Java que valida los valores de los parámetros. La estructura de este bloque es:

```
requires { expression }
```

Los bloques *java*, *ansi\_c* y *cplusplus* describen el comportamiento de la función por medio de su descripción como el cuerpo de una función en los lenguajes de programación Java, C y C++, respectivamente. Las variables de entrada para estas funciones son 'a' y 'b'. El formato de estos bloques es el siguiente:

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

El bloque *derivative* describe la derivada de la función con respecto a las variables de entrada 'a' y 'b'. Esta descripción consiste en una expresión Java de asignamiento a la variable '*deriv*[]'. La derivada de la función con respecto a la variable de entrada 'a' debe ser asignada a '*deriv*[0]', mientras que la derivada de la función con respecto a la variable de entrada 'b' debe ser asignada a '*deriv*[1]'. La descripción de la derivada de la función permite propagar la derivada de la función de error del sistema utilizada por los algoritmos de aprendizaje supervisado basados en gradiente descendente. El formato es:

```
derivative { Java_expressions }
```

El bloque *source* es utilizado para definir código Java que es directamente incluido en el código de la clase generada para la definición de la función. Este código nos permite definir métodos locales que pueden ser empleados dentro de otros bloques. La estructura es:

```
source { Java_code }
```



El siguiente ejemplo muestra la definición de la T-norma mínimo, también usada como función de implicación de Mamdani.

```
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));
  }
}
```

## Definición de funciones unarias

Las funciones unarias son usadas para describir modificadores lingüísticos. Estas funciones pueden ser asignadas a los modificadores no (*not*), fuertemente (*strongly*), más o menos (*more-or-less*) y ligeramente (*slightly*). La estructura de la definición de una función unaria es como sigue:

```
unary identifier { blocks }
```

Los bloques que pueden aparecer en la definición de una función unaria son: *alias*, *parameter*, *requires*, *java*, *ansi\_c*, *cplusplus*, *derivative* y *source*.

El bloque *alias* se utiliza para definir nombres alternativos para identificar a la función. Cualquiera de esos identificadores puede ser usado para hacer referencia a la función. La sintaxis del bloque *alias* es:

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

El bloque *parameter* permite la definición de los parámetros de los que depende la función. Su formato es:

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

El bloque *requires* expresa las restricciones sobre los valores de los parámetros por medio de una expresión Booleana en Java que valida los valores de los parámetros. La estructura de este bloque es:

```
requires { expression }
```

Los bloques *java*, *ansi\_c* y *cplusplus* describen el comportamiento de la función por medio de su descripción como el cuerpo de una función en los lenguajes de programación Java, C y C++, respectivamente. La variable de entrada para estas funciones es 'a'. El formato de estos bloques es el siguiente:

```
java { Java_function_body }
ansi_c { C_function_body }
cplusplus { Cplusplus_function_body }
```

El bloque *derivative* describe la derivada de la función con respecto a la variable de entrada 'a'. Esta descripción consiste en una expresión Java de asignamiento a la variable '*deriv*'. La descripción de la derivada de la función permite propagar la derivada de la función de error del sistema utilizada por los algoritmos de aprendizaje supervisado basados en gradiente descendente. El formato es:

```
derivative { Java_expressions }
```

El bloque *source* es utilizado para definir código Java que es directamente incluido en el código de la clase generada para la definición de la función. Este código nos permite definir métodos locales que pueden ser empleados dentro de otros bloques. La estructura es:

```
source { Java_code }
```

El siguiente ejemplo muestra la definición de la C-norma de Yager, que depende del parámetro *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 ); }
}
```

## Definición de funciones no difusas

Las funciones no difusas se utilizan para describir operaciones matemáticas entre variables con valores no difusos. Estas funciones pueden ser asignadas a bloques no difusos que pueden incluirse en la definición de la jerarquía modular de los sistemas difusos. La estructura de la definición de una función no difusa es como sigue:

```
crisp identifier { blocks }
```

Los bloques que pueden aparecer en la definición de una función no difusa son: *alias*, *parameter*, *requires*, *inputs*, *java*, *ansi\_c*, *cplusplus* y *source*.

El bloque *alias* se utiliza para definir nombres alternativos para identificar a la función. Cualquiera de esos identificadores puede ser usado para hacer referencia a la función. La sintaxis del bloque *alias* es:

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

El bloque *parameter* permite la definición de los parámetros de los que depende la función. El último identificador puede ir seguido de corchetes para definir una lista de parámetros. Su formato es:

```
parameter identifier, identifier, ..., identifier[] ;
```

El bloque *requires* expresa las restricciones sobre los valores de los parámetros por medio de una expresión Booleana en Java que valida los valores de los parámetros. La estructura de este bloque es:

```
requires { expression }
```

El bloque *inputs* define el número de variables de entrada de la función no difusa por medio de una expresión Java que devuelve un valor entero. La sintaxis de este bloque es:

```
inputs { Java_function_body }
```

Los bloques *java*, *ansi\_c* y *cplusplus* describen el comportamiento de la función por medio de su descripción como el cuerpo de una función en los lenguajes de programación Java, C y C++, respectivamente. La variable 'x[]' contiene los valores de la variable de entrada. El formato de estos bloques es el siguiente:

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

El bloque *source* es utilizado para definir código Java que es directamente incluido en el código de la clase generada para la definición de la función. Este código nos permite definir métodos locales que pueden ser empleados dentro de otros bloques. La estructura es:

```
source { Java_code }
```

El siguiente ejemplo muestra la definición de una función que suma N variables de entrada.

```
crisp addN {
  parameter N;
  requires { N>0 }
  inputs { return (int) N; }
  java {
    double a = 0;
    for(int i=0; i<N; i++) a+=x[i];
    return a;
  }
  ansi_c {
    int i;
    double a = 0;
    for(i=0; i<N; i++) a+=x[i];
    return a;
  }
  cplusplus {
    double a = 0;
    for(int i=0; i<N; i++) a+=x[i];
    return a;
  }
}
```

## Definición de funciones de pertenencia

Las funciones de pertenencia son asignadas a las etiquetas lingüísticas que forman un tipo de variable lingüística. La estructura de una definición de función de pertenencia en un paquete de funciones es como sigue:

```
mf identifier { blocks }
```

Los bloques que pueden aparecer en la definición de una función de pertenencia son: *alias*, *parameter*, *requires*, *java*, *ansi\_c*, *cplusplus*, *derivative*, *update* y *source*.

El bloque *alias* se utiliza para definir nombres alternativos para identificar a la función. Cualquiera de esos identificadores puede ser usado para hacer referencia a la función. La sintaxis del bloque *alias* es:

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

El bloque *parameter* permite la definición de los parámetros de los que depende la función. El último identificador puede ir seguido de corchetes para definir una lista de parámetros. Su formato es:

```
parameter identifier, identifier, ..., identifier[] ;
```

El bloque *requires* expresa las restricciones sobre los valores de los parámetros por medio de una expresión Booleana en Java que valida los valores de los parámetros. Esta expresión puede usar también los valores de las variables '*min*' y '*max*', que representan los valores mínimo y máximo del universo de discurso de la variable lingüística considerada. La estructura de este bloque es:

```
requires { expression }
```

Los bloques *java*, *ansi\_c* y *cplusplus* describen el comportamiento de la función por medio de su descripción como el cuerpo de una función en los lenguajes de programación Java, C y C++, respectivamente. El formato de estos bloques es el siguiente:

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

La definición de una función de pertenencia incluye no sólo la descripción del comportamiento de la función en sí misma, sino también del comportamiento de la función bajo la acción de los modificadores *greater-or-equal* y *smaller-or-equal*, así como el cálculo de los valores del centro y la base de la función de pertenencia. Como consecuencia, los bloques *java*, *ansi\_c* y *cplusplus* se dividen en los siguientes subbloques:

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

El subbloque *equal* describe el comportamiento de la función. Los subbloques *greatereq* y *smallereq* describen la acción de los modificadores greater-or-equal y smaller-or-equal respectivamente. La variable de entrada en estos subbloques se denomina 'x'. El código puede usar los valores de los parámetros de la función, así como las variables 'min' y 'max', que representan los valores mínimo y máximo del universo de discurso de la función. Los subbloques *greatereq* y *smallereq* pueden ser omitidos. En ese caso las transformaciones correspondientes son calculadas recorriendo todos los valores del universo de discurso. Sin embargo, resulta mucho más eficiente usar la función analítica, por lo que la definición de estos subbloques está fuertemente recomendada.

Los subbloques *center* y *basis* describen el centro y la base de la función de pertenencia. El código de estos subbloques puede usar los valores de los parámetros de la función y las variables 'min' y 'max'. Esta información es usada por varios métodos de defuzzificación simplificados. Estos subbloques son opcionales y su función por defecto devuelve un valor nulo.

El bloque *derivative* describe la derivada de la función con respecto a cada parámetro. Este bloque es también dividido en los subbloques *equal*, *greatereq*, *smallereq*, *center* y *basis*. El código de estos subbloques consiste en expresiones Java que asignan valores a la variable 'deriv[]'. El valor de 'deriv[i]' representa la derivada de la función con respecto al i-ésimo parámetro de la función de pertenencia. La descripción de la derivada de la función permite calcular la derivada de la función de error del sistema utilizada por los algoritmos de aprendizaje basados en gradiente descendente. El formato es:

```
derivative { subblocks }
```

El bloque *update* se utiliza para calcular unos valores válidos para el conjunto de parámetros (almacenados en la variable *pos[]*) a partir de los desplazamientos deseados (almacenados en la variable *disp[]*) generados en un proceso de ajuste automático, teniendo en cuenta la selección de parámetros a ajustar (almacenados en la variable booleana *adj[]*). Un tipo de restricción muy común para el desplazamiento consiste en mantener el orden de los parámetros. Este proceso puede garantizarse utilizando la función *sortedUpdate(pos,disp,adj)*. El código Java puede utilizar las variables 'min', 'max' y 'step', que representan el mínimo, el máximo y la división del universo de discurso, respectivamente. La sintaxis del bloque *update* es:

```
update { Java_function_body }
```

El bloque *source* es utilizado para definir código Java que es directamente incluido en el código de la clase generada para la definición de la función. Este código nos permite definir métodos locales que pueden ser empleados dentro de otros bloques. La estructura es:

```
source { Java_code }
```

El siguiente ejemplo muestra la definición de una función de pertenencia en forma de triángulo.

```

mf triangle {
  parameter a, b, c;
  requires { a<b && b<c && b>=min && b<=max }
  java {
    equal { return (a<x && x<=b? (x-a)/(b-a) : (b<x && x<c? (c-x)/(c-
b) : 0)); }
    greatereq { return (x<a? 0 : (x>b? 1 : (x-a)/(b-a) )); }
    smallereq { return (x<b? 1 : (x>c? 0 : (c-x)/(c-b) )); }
    center { return b; }
    basis { return (c-a); }
  }
  ansi_c {
    equal { return (a<x && x<=b? (x-a)/(b-a) : (b<x && x<c? (c-x)/(c-
b) : 0)); }
    greatereq { return (x<a? 0 : (x>b? 1 : (x-a)/(b-a) )); }
    smallereq { return (x<b? 1 : (x>c? 0 : (c-x)/(c-b) )); }
    center { return b; }
    basis { return (c-a); }
  }
  cplusplus {
    equal { return (a<x && x<=b? (x-a)/(b-a) : (b<x && x<c? (c-x)/(c-
b) : 0)); }
    greatereq { return (x<a? 0 : (x>b? 1 : (x-a)/(b-a) )); }
    smallereq { return (x<b? 1 : (x>c? 0 : (c-x)/(c-b) )); }
    center { return b; }
    basis { return (c-a); }
  }
  derivative {
    equal {
      deriv[0] = (a<x && x<b ? (x-b)/((b-a)*(b-a)) : (x==a? 0.5/(a-b) :
0));
      deriv[1] = (a<x && x<b ? (a-x)/((b-a)*(b-a)) :
(b<x && x<c ? (c-x)/((c-b)*(c-b)) :
(x==b? 0.5/(a-b) + 0.5/(c-b) : 0));
      deriv[2] = (b<x && x<c ? (x-b)/((c-b)*(c-b)) : (x==c? 0.5/(c-b) :
0));
    }
    greatereq {
      deriv[0] = (a<x && x<b ? (x-b)/((b-a)*(b-a)) : (x==a? 0.5/(a-b) :
0));
      deriv[1] = (a<x && x<b ? (a-x)/((b-a)*(b-a)) : (x==b? 0.5/(a-b) :
0));
      deriv[2] = 0;
    }
    smallereq {
      deriv[0] = 0;
      deriv[1] = (b<x && x<c ? (c-x)/((c-b)*(c-b)) : (x==b? 0.5/(c-b) :
0));
      deriv[2] = (b<x && x<c ? (x-b)/((c-b)*(c-b)) : (x==c? 0.5/(c-b) :
0));
    }
    center {
      deriv[0] = 1;
      deriv[1] = 1;
      deriv[2] = 1;
    }
    basis {
      deriv[0] = -1;
      deriv[1] = 0;
      deriv[2] = 1;
    }
  }
}

```

```

    }
  }
  update {
    pos = sortedUpdate(pos, desp, adj);
    if(pos[1]<min) pos[1]=min;
    if(pos[2]<=pos[1]) pos[2] = pos[1]+step;
    if(pos[1]>max) pos[1]=max;
    if(pos[0]>=pos[1]) pos[0] = pos[1]-step;
  }
}

```

## Definición de familias de funciones de pertenencia

Las familias de funciones de pertenencia describen conjuntos de funciones de pertenencia que comparten una lista de parámetros. Las familias se utilizan para definir conjuntos de funciones de pertenencia con unas ciertas restricciones, como la simetría entre las funciones, el orden entre ellas o un grado de solapamiento fijo. Cada función de pertenencia es referenciada por medio de su índice dentro de la familia. La estructura de una definición de una familia de funciones de pertenencia en un paquete de funciones es como sigue:

```
family identifier { blocks }
```

Los bloques que pueden aparecer en la definición de una familia de funciones de pertenencia son: *alias*, *parameter*, *requires*, *members*, *java*, *ansi\_c*, *cplusplus*, *derivative*, *update* y *source*.

El bloque *alias* se utiliza para definir nombres alternativos para identificar a la familia. Cualquiera de esos identificadores puede ser usado para hacer referencia a la familia. La sintaxis del bloque *alias* es:

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

El bloque *parameter* permite la definición de los parámetros de los que depende la familia. El último identificador puede ir seguido de corchetes para definir una lista de parámetros. Su formato es:

```
parameter identifier, identifier, ..., identifier[] ;
```

El bloque *requires* expresa las restricciones sobre los valores de los parámetros por medio de una expresión Booleana en Java que valida los valores de los parámetros. Esta expresión puede usar también los valores de las variables '*min*' y '*max*', que representan los valores mínimo y máximo del universo de discurso de la variable lingüística considerada. La estructura de este bloque es:

```
requires { expression }
```

El bloque *members* define el número de funciones de pertenencia incluidas en la familia por medio de una expresión en Java que devuelve un valor entero. La sintaxis de este bloque es:

```
members { Java_function_body }
```

Los bloques *java*, *ansi\_c* y *cplusplus* describen el comportamiento de la función por medio de su descripción como el cuerpo de una función en los lenguajes de programación Java, C y C++, respectivamente. El formato de estos bloques es el siguiente:

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

La definición de una familia de funciones de pertenencia incluye no sólo la descripción del comportamiento de la función en sí misma, sino también del comportamiento de las funciones bajo la acción de los modificadores *greater-or-equal* y *smaller-or-equal*, así como el cálculo de los valores del centro y la base de las funciones de pertenencia. Como consecuencia, los bloques *java*, *ansi\_c* y *cplusplus* se dividen en los siguientes subbloques:

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

El subbloque *equal* describe el comportamiento de la función. Los subbloques *greatereq* y *smallereq* describen la acción de los modificadores *greater-or-equal* y *smaller-or-equal* respectivamente. La variable '*i*' contiene el índice que permite identificar la función de pertenencia de la familia. La variable de entrada en estos subbloques se denomina '*x*'. El código puede usar los valores de los parámetros de la función, así como las variables '*min*' y '*max*', que representan los valores mínimo y máximo del universo de discurso de la familia. Los subbloques *greatereq* y *smallereq* pueden ser omitidos. En ese caso las transformaciones correspondientes son calculadas recorriendo todos los valores del universo de discurso. Sin embargo, resulta mucho más eficiente usar la función analítica, por lo que la definición de estos subbloques está fuertemente recomendada.

Los subbloques *center* y *basis* describen el centro y la base de las funciones de pertenencia. El código de estos subbloques puede usar los valores de la variable '*i*' (el índice de la función de pertenencia), los parámetros de la familia y las variables '*min*' y '*max*'. Esta información es usada por varios métodos de defuzzificación simplificados. Estos subbloques son opcionales y su función por defecto devuelve un valor nulo.

El bloque *derivative* describe la derivada de cada función con respecto a cada parámetro. Este bloque es también dividido en los subbloques *equal*, *greatereq*, *smallereq*, *center* y *basis*. El código de estos subbloques consiste en expresiones Java que asignan valores a la variable '*deriv[]*'. El valor de '*deriv[i]*' representa la derivada de cada función con respecto al *i*-ésimo parámetro de la familia. La descripción de la derivada de la función permite calcular la derivada de la función de error del sistema utilizada por los algoritmos de aprendizaje basados en gradiente descendente. El formato es:

```
derivative { subblocks }
```

El bloque *update* se utiliza para calcular unos valores válidos para el conjunto de parámetros (almacenados en la variable *pos[]*) a partir de los desplazamientos deseados (almacenados en la variable *disp[]*) generados en un proceso de ajuste automático, teniendo en cuenta la selección de parámetros a ajustar (almacenados en la variable booleana *adj[]*). Un tipo de restricción muy común para el desplazamiento consiste en mantener el orden de los parámetros. Este proceso puede garantizarse utilizando la función *sortedUpdate(pos,disp,adj)*.



El código Java puede utilizar las variables '*min*', '*max*' y '*step*', que representan el mínimo, el máximo y la división del universo de discurso, respectivamente. La sintaxis del bloque *update* es:

```
update { Java_function_body }
```

El bloque *source* es utilizado para definir código Java que es directamente incluido en el código de la clase generada para la definición de la función. Este código nos permite definir métodos locales que pueden ser empleados dentro de otros bloques. La estructura es:

```
source { Java_code }
```

El siguiente ejemplo muestra la definición de la familia de funciones de pertenencia *triangular*.

```
family triangular {
  parameter p[];
  requires { p.length==0 || (p.length>0 && p[0]>min && p[p.length-
1]<max && sorted(p)) }
  members { return p.length+2; }
  java {
    equal {
      double a = (i==0? min-1 : (i==1 ? min : p[i-2]));
      double b = (i==0? min : (i==p.length+1? max : p[i-1]));
      double c = (i==p.length? max : (i==p.length+1? max+1 : p[i]));
      return (a<x && x<=b? (x-a)/(b-a) : (b<x && x<c? (c-x)/(c-b): 0));
    }
    greatereq {
      double a = (i==0? min-1 : (i==1 ? min : p[i-2]));
      double b = (i==0? min : (i==p.length+1? max : p[i-1]));
      return (x<a? 0 : (x>b? 1 : (x-a)/(b-a) ));
    }
    smallereq {
      double b = (i==0? min : (i==p.length+1? max : p[i-1]));
      double c = (i==p.length? max : (i==p.length+1? max+1 : p[i]));
      return (x<b? 1 : (x>c? 0 : (c-x)/(c-b) ));
    }
    center {
      double b = (i==0? min : (i==p.length+1? max : p[i-1]));
      return b;
    }
    basis {
      double a = (i<=1 ? min : p[i-2]);
      double c = (i>=p.length? max : p[i]);
      return (c-a);
    }
  }
  ansi_c {
    equal {
      double a = (i==0? min-1 : (i==1 ? min : p[i-2]));
      double b = (i==0? min : (i==length+1? max : p[i-1]));
      double c = (i==length? max : (i==length+1? max+1 : p[i]));
      return (a<x && x<=b? (x-a)/(b-a) : (b<x && x<c? (c-x)/(c-b): 0));
    }
    greatereq {
      double a = (i==0? min-1 : (i==1 ? min : p[i-2]));
      double b = (i==0? min : (i==length+1? max : p[i-1]));
      return (x<a? 0 : (x>b? 1 : (x-a)/(b-a) ));
    }
  }
}
```

```

smallereq {
  double b = (i==0? min : (i==length+1? max : p[i-1]));
  double c = (i==length? max : (i==length+1? max+1 : p[i]));
  return (x<b? 1 : (x>c? 0 : (c-x)/(c-b) ));
}
center {
  double b = (i==0? min : (i==length+1? max : p[i-1]));
  return b;
}
basis {
  double a = (i<=1 ? min : p[i-2]);
  double c = (i>=length? max : p[i]);
  return (c-a);
}
}
cplusplus {
  equal {
    double a = (i==0? min-1 : (i==1 ? min : p[i-2]));
    double b = (i==0? min : (i==length+1? max : p[i-1]));
    double c = (i==length? max : (i==length+1? max+1 : p[i]));
    return (a<x && x<=b? (x-a)/(b-a) : (b<x && x<c? (c-x)/(c-b) : 0));
  }
  greatereq {
    double a = (i==0? min-1 : (i==1 ? min : p[i-2]));
    double b = (i==0? min : (i==length+1? max : p[i-1]));
    return (x<a? 0 : (x>b? 1 : (x-a)/(b-a) ));
  }
  smallereq {
    double b = (i==0? min : (i==length+1? max : p[i-1]));
    double c = (i==length? max : (i==length+1? max+1 : p[i]));
    return (x<b? 1 : (x>c? 0 : (c-x)/(c-b) ));
  }
  center {
    double b = (i==0? min : (i==length+1? max : p[i-1]));
    return b;
  }
  basis {
    double a = (i<=1 ? min : p[i-2]);
    double c = (i>=length? max : p[i]);
    return (c-a);
  }
}
}
derivative {
  equal {
    double a = (i==0? min-1 : (i==1 ? min : p[i-2]));
    double b = (i==0? min : (i==p.length+1? max : p[i-1]));
    double c = (i==p.length? max : (i==p.length+1? max+1 : p[i]));
    if(i>=2) {
      if(a<x && x<b) deriv[i-2] = (x-b)/((b-a)*(b-a));
      else if(x==a) deriv[i-2] = 0.5/(a-b);
      else deriv[i-2] = 0;
    }
    if(i>=1 && i<=p.length) {
      if(a<x && x<b) deriv[i-1] = (a-x)/((b-a)*(b-a));
      else if(b<x && x<c) deriv[i-1] = (c-x)/((c-b)*(c-b));
      else if(x==b) deriv[i-1] = 0.5/(a-b) + 0.5/(c-b);
      else deriv[i-1] = 0;
    }
    if(i<p.length) {
      if(b<x && x<c) deriv[i] = (x-b)/((c-b)*(c-b));
    }
  }
}

```

```

    else if(x==c) deriv[i] = 0.5/(c-b);
    else deriv[i] = 0;
  }
}
greater {
  double a = (i==0? min-1 : (i==1 ? min : p[i-2]));
  double b = (i==0? min : (i==p.length+1? max : p[i-1]));
  if(i>=2) {
    if(a<x && x<b) deriv[i-2] = (x-b)/((b-a)*(b-a));
    else if(x==a) deriv[i-2] = 0.5/(a-b);
    else deriv[i-2] = 0;
  }
  if(i>=1 && i<=p.length) {
    if(a<x && x<b) deriv[i-1] = (a-x)/((b-a)*(b-a));
    else if(x==b) deriv[i-1] = 0.5/(a-b);
    else deriv[i-1] = 0;
  }
}
smaller {
  double b = (i==0? min : (i==p.length+1? max : p[i-1]));
  double c = (i==p.length? max : (i==p.length+1? max+1 : p[i]));
  if(i>=1 && i<=p.length) {
    if(b<x && x<c) deriv[i-1] = (c-x)/((c-b)*(c-b));
    else if(x==b) deriv[i-1] = 0.5/(c-b);
    else deriv[i-1] = 0;
  }
  if(i<p.length) {
    if(b<x && x<c) deriv[i] = (x-b)/((c-b)*(c-b));
    else if(x==c) deriv[i] = 0.5/(c-b);
    else deriv[i] = 0;
  }
}
center {
  if(i>=1 && i<=p.length) deriv[i-1] = 1;
}
basis {
  if(i>1) deriv[i-2] = -1;
  if(i<p.length) deriv[i] = 1;
}
}
update {
  if(p.length == 0) return;
  pos = sortedUpdate(pos, desp, adj);
  if(pos[0]<=min) {
    pos[0]=min+step;
    for(int i=1; i<p.length; i++) {
      if(pos[i]<=pos[i-1]) pos[i] = pos[i-1]+step;
      else break;
    }
  }
  if(pos[p.length-1]>=max) {
    pos[p.length-1]=max-step;
    for(int i=p.length-2; i>=0; i--) {
      if(pos[i]>=pos[i+1]) pos[i] = pos[i+1]-step;
      else break;
    }
  }
}
}
}
}

```

## Definición de métodos de defuzzificación

Los métodos de defuzzificación obtienen el valor representativo de un conjunto difuso. Estos métodos son utilizados en la etapa final del proceso de inferencia difuso cuando no es posible trabajar con conclusiones difusas. La estructura de una definición de método de defuzzificación en un paquete de funciones es como sigue:

```
defuz identifier { blocks }
```

Los bloques que pueden aparecer en una definición de método de defuzzificación son: *alias*, *parameter*, *requires*, *definedfor*, *java*, *ansi\_c*, *cplusplus* y *source*.

El bloque *alias* se utiliza para definir nombres alternativos para identificar al método. Cualquiera de esos identificadores puede ser usado para hacer referencia al método. La sintaxis del bloque *alias* es:

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

El bloque *parameter* permite la definición de los parámetros de los que depende el método. Su formato es:

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

El bloque *requires* expresa las restricciones sobre los valores de los parámetros por medio de una expresión Booleana en Java que valida los valores de los parámetros. La estructura de este bloque es:

```
requires { expression }
```

El bloque *definedfor* se utiliza para enumerar los tipos de funciones de pertenencia que el método puede usar como conclusiones parciales. Este bloque ha sido incluido porque algunos métodos de defuzzificación simplificados solamente trabajan con ciertas funciones de pertenencia. Este bloque es opcional. Por defecto, se asume que el método puede trabajar con todas las funciones de pertenencia. La estructura del bloque es:

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

El bloque *source* es utilizado para definir código Java que es directamente incluido en el código de la clase generada para la definición del método. Este código nos permite definir funciones locales que pueden ser empleadas dentro de otros bloques. La estructura es:

```
source { Java_code }
```

Los bloques *java*, *ansi\_c* y *cplusplus* describen el comportamiento del método por medio de su descripción como el cuerpo de una función en los lenguajes de programación Java, C y C++, respectivamente. El formato de estos bloques es el siguiente:

```
java { Java_function_body }
ansi_c { C_function_body }
cplusplus { Cplusplus_function_body }
```

La variable de entrada para estas funciones es el objeto '*mf*', que encapsula al conjunto difuso obtenido como conclusión del proceso de inferencia. El código puede usar el valor de las variables '*min*', '*max*' y '*step*', que representan respectivamente el mínimo, el máximo y la

división del universo de discurso del conjunto difuso. Los métodos de defuzzificación convencionales se basan en recorrer todos los valores del universo de discurso calculando el grado de pertenencia para cada valor del universo. Por otra parte, los métodos de defuzzificación simplificados suelen recorrer las conclusiones parciales calculando el valor representativo en términos de los grados de activación, centros, bases y parámetros de estas conclusiones parciales. Como se muestra en la siguiente tabla, el modo en que dicha información es accedida por el objeto *mf* depende del lenguaje de programación utilizado.

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

El siguiente ejemplo muestra la definición del método clásico de defuzzificación del centro de área.

```
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;
    }
}

```

El siguiente ejemplo muestra la definición de un método de defuzzificación simplificado, la media difusa ponderada (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;
    }
}

```

Este último ejemplo muestra la definición del método de Takagi-Sugeno de primer orden.

```

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;
  }
}
}

```

## El paquete estándar xfl

El lenguaje de especificación XFL3 permite al usuario definir sus propias funciones de pertenencia, familias de funciones de pertenencia, funciones no difusas, métodos de defuzzificación y funciones relacionadas con las conectivas difusas y los modificadores lingüísticos. Con objeto de facilitar el uso de XFL3, las funciones más conocidas han sido incluidas en un paquete estándar llamado *xfl*. Las funciones binarias incluidas son las siguientes:

Nombre	Tipo	Descripción Java
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	$(a <= b ? 1 : 0)$

Las funciones unarias incluidas en el paquete *xfl* son:

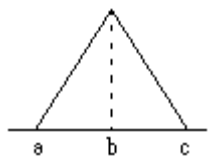
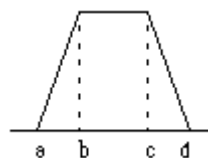
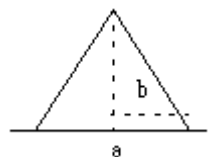
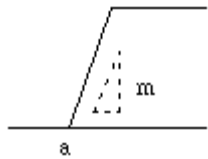
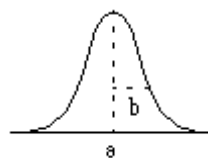
Nombre	Parámetro	Descripción Java
not	-	$(1 - a)$
sugeno	l	$(1 - a) / (1 + a * l)$
square	-	$(a * a)$
cubic	-	$(a * a * a)$
sqrt	-	$\text{Math.sqrt}(a)$
yager	w	$\text{Math.pow}( ( 1 - \text{Math.pow}(a, w) ) , 1 / w )$
pow	w	$\text{Math.pow}(a, w)$
parabola	-	$4 * a * (1 - a)$
edge	-	$(a <= 0.5 ? 2 * a : 2 * (1 - a))$

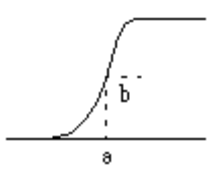
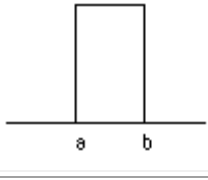
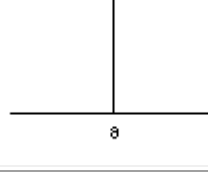


Las funciones no difusas incluidas en el paquete *xfl* son:


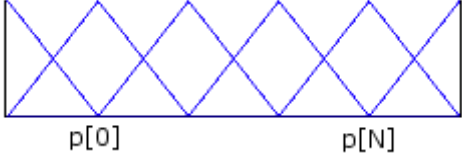
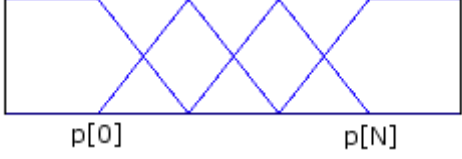

Nombre	Parámetro	Descripción
add2	-	Suma de variables
addN	N	Suma de N variables
addDeg	-	Suma de dos variables angulares (en grados)
addRad	-	Suma de dos variables angulares (en radianes)
diff2	-	Diferencia entre dos variables
diffDeg	-	Diferencia entre dos variables angulares (en grados)
diffRad	-	Diferencia entre dos variables angulares (en radianes)
prod	-	Producto de dos variables
div	-	División entre dos variables
select	N	Selección entre N variables

Las funciones de pertenencia definidas en el paquete *xfl* son las siguientes:

Nombre	Parámetros	Descripción
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	-

Las familias de funciones de pertenencia definidas en el paquete *xfi* son las siguientes:

Nombre	Parámetros	Descripción
rectangular	p[]	
triangular	p[]	
sh_triangular	p[]	
spline	p[]	

Los métodos de defuzzificación definidos en el paquete estándar son:

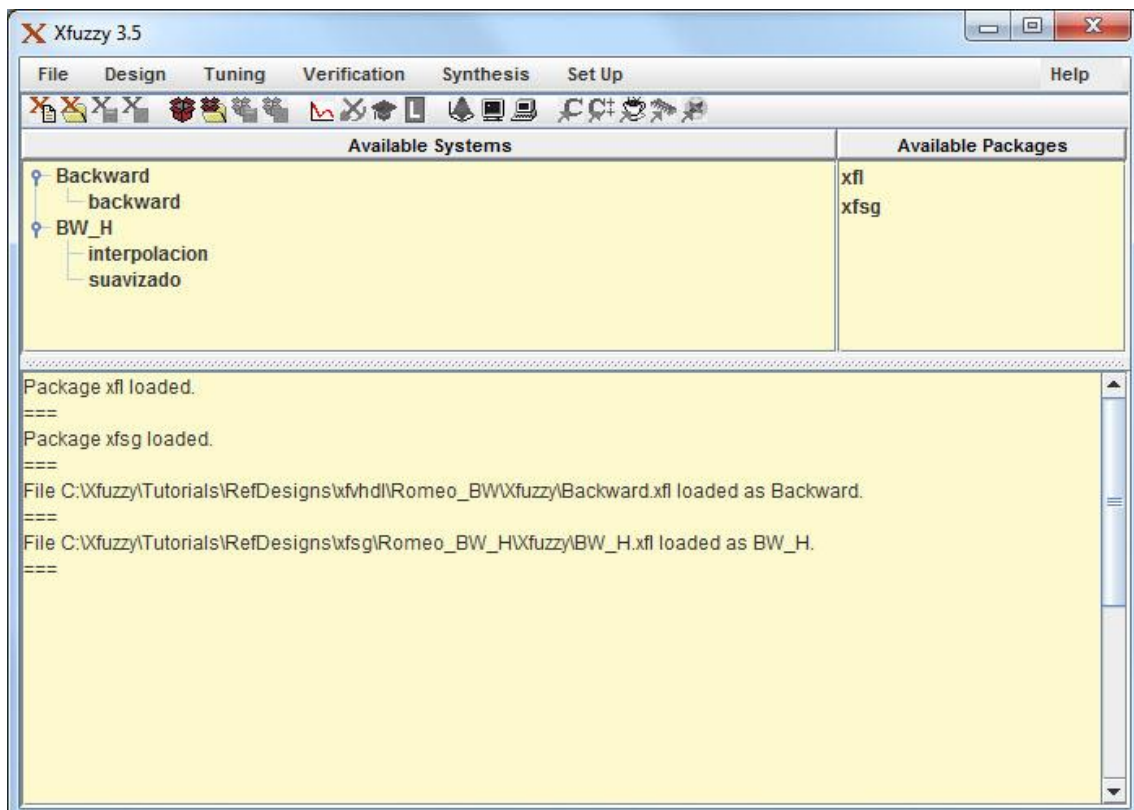
<b>Nombre</b>	<b>Tipo</b>	<b>Definido para</b>
CenterOfArea	Conventional	any function
FirstOfMaxima	Conventional	any function
LastOfMaxima	Conventional	any function
MeanOfMaxima	Conventional	any function
FuzzyMean	Simplified	triangle, isosceles, trapezoid, bell, rectangle, singleton
WeightedFuzzyMean	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



## Entorno de desarrollo Xfuzzy 3

- Entorno de desarrollo Xfuzzy 3
  - Etapa de [descripción](#)
    - Edición de sistemas ([xfedit](#))
    - Edición de paquetes ([xfpkg](#))
  - Etapa de [verificación](#)
    - Representación gráfica ([xfplot](#))
    - Monitor de inferencias ([xfmt](#))
    - Simulación de sistemas ([xfsim](#))
  - Etapa de [ajuste](#)
    - Adquisición de conocimiento ([xfdm](#))
    - Predicción de series temporales ([xftsp](#))
    - Aprendizaje supervisado ([xfsl](#))
    - Simplificación ([xfsp](#))
  - Etapa de [síntesis](#)
    - Generador de código C ([xfc](#))
    - Generador de código C++ ([xfcpp](#))
    - Generador de código Java ([xfj](#))
    - Generador de código VHDL ([xfvhdl](#))
    - Generador de modelo SysGen ([xfsg](#))

*Xfuzzy 3* es un entorno de desarrollo de sistemas difusos que incluye herramientas de CAD que cubren las diferentes etapas de diseño. El entorno integra las distintas herramientas bajo una interfaz gráfica de usuario que facilita el proceso de diseño. La siguiente figura muestra la ventana principal del entorno.



La barra de menús en la ventana principal contiene los enlaces a las diferentes herramientas. Bajo la barra de menús se sitúa una barra de botones con las opciones más utilizadas. La zona central de la ventana muestra dos listas. La primera es la lista de sistemas cargados (el entorno puede trabajar con varios sistemas simultáneamente). Esta lista muestra las especificaciones mediante una estructura desplegable, de forma que es posible seleccionar el sistema completo o cualquiera de sus bases de reglas como la especificación activa sobre la que actuarán las distintas herramientas. La segunda lista contiene los paquetes cargados. El resto de la ventana principal está ocupado por un área de mensajes.

La barra de menús está dividida en las diferentes etapas del desarrollo de un sistema: El menú **File** permite crear (*create*), cargar (*load*), salvar (*save*) y cerrar (*close*) un sistema difuso. Este menú contiene también las opciones para crear, cargar, salvar y cerrar un paquete de funciones. El menú termina con la opción para salir del entorno. El menú **Design** se utiliza para editar el sistema difuso seleccionado ([xfedit](#)) o el paquete de funciones (*package*) seleccionado ([xfpkg](#)). El menú **Tuning** contiene los enlaces a la herramienta de adquisición de conocimiento ([xfdm](#)), la herramienta de predicción de series temporales ([xftsp](#)), la herramienta de aprendizaje supervisado ([xfsl](#)) y la herramienta de simplificación ([xfsp](#)). El menú **Verification** permite representar el comportamiento del sistema mediante una gráfica bidimensional o tridimensional ([xfplot](#)), monitorizar el sistema ([xfmt](#)) y simularlo en combinación con un modelo Java de su entorno de operación ([xfsim](#)). El menú **Synthesis** está dividido en dos partes: la síntesis software, que genera descripciones del sistema en C ([xfc](#)), C++ ([xfcpp](#)) y Java ([xfj](#)); y la síntesis hardware que traslada la descripción de un sistema difuso a código VHDL ([xfvhdl](#)) o a un modelo Simulink para la herramienta SysGen de Xilinx ([xfsg](#)). El menú **Set Up** se utiliza para modificar el directorio de trabajo del entorno, salvar los mensajes del entorno en un fichero de log externo, cerrar el fichero de log, limpiar el área de mensajes de la ventana principal y cambiar la apariencia (*look & feel*) del entorno.

Muchas opciones de la barra de menús sólo están activas cuando se selecciona un sistema difuso o una base de reglas. Para seleccionar un sistema difuso o una base de reglas basta con pulsar sobre su nombre en la lista de sistemas. Una doble pulsación sobre el nombre abrirá la herramienta de edición. El mismo resultado se obtiene presionando la tecla *Enter* una vez que el elemento ha sido seleccionado. La tecla *Insert* creará un nuevo sistema y la tecla *Delete* se utiliza para cerrar el sistema. Estos aceleradores son comunes a todas las listas del entorno: *Insert* se utiliza para insertar un nuevo elemento a la lista; *Enter* o una doble pulsación editará el elemento seleccionado; y *Delete* quitará el elemento de la lista.

## Etapa de descripción

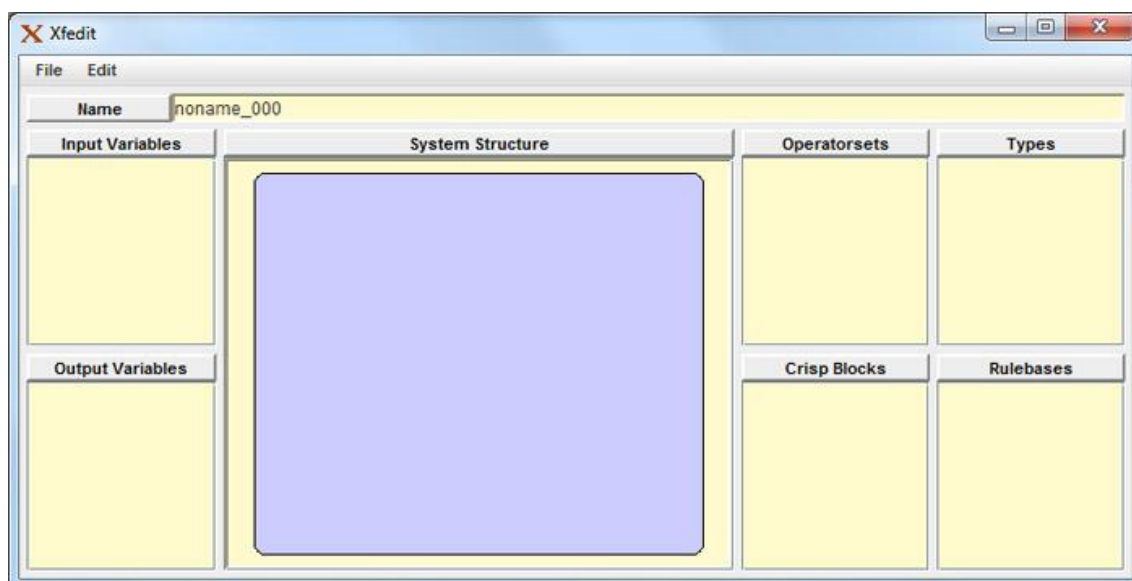
El primer paso en el desarrollo de un sistema difuso consiste en seleccionar una descripción preliminar del sistema. Esta descripción será posteriormente refinada como resultado de las etapas de ajuste y verificación.

*Xfuzzy 3* contiene dos herramientas que facilitan la descripción de sistemas difusos: [xfedit](#) y [xfpkg](#). La primera está dedicada a la definición lógica del sistema, es decir, la definición de sus variables lingüísticas y las relaciones lógicas entre ellas. Por otra parte, la herramienta [xfpkg](#) facilita la descripción de las funciones matemáticas asignadas a los operadores difusos, los modificadores lingüísticos, las funciones de pertenencia y los métodos de defuzzificación.

## Herramienta de edición de sistemas – Xfedit

La herramienta *xfedit* proporciona una interfaz gráfica para facilitar la descripción de sistemas difusos, evitando al usuario la necesidad de conocer en profundidad el lenguaje XFL3. La herramienta está formada por un conjunto de ventanas que permiten al usuario crear y editar los conjuntos de operadores, los tipos de variables lingüísticas y las bases de reglas incluidas en el sistema difuso, así como describir la estructura jerárquica del sistema bajo desarrollo.

La herramienta puede ser ejecutada directamente desde la línea de comandos con la expresión "*xfedit file.xfl*", o desde la ventana principal del entorno usando la opción *System Edition* del menú *Design*.

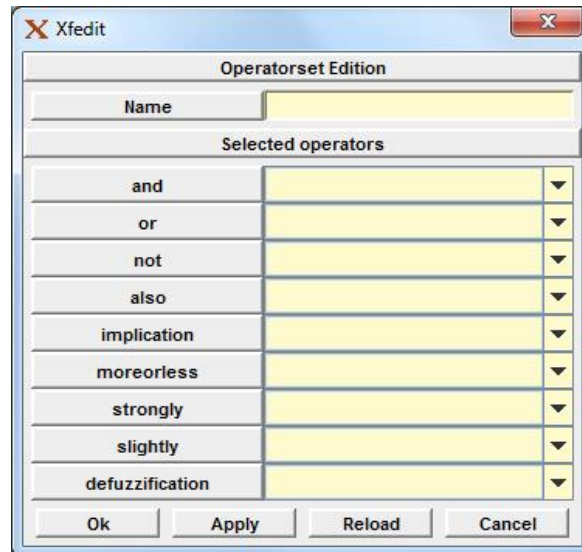


La figura muestra la ventana principal de *xfedit*. El menú *File* contiene las siguientes opciones: "*Save*", "*Save As*", "*Edit XFL3 File*" y "*Close Edition*". Las opciones "*Save*" y "*Save As*" se utilizan para salvar el estado actual de la definición del sistema. La opción "*Edit XFL3 File*" abre una ventana de texto para editar la descripción XFL3 del sistema. La última opción del menú se emplea para cerrar la herramienta. El campo *Name* bajo la barra de menús no es editable. El nombre del sistema bajo desarrollo puede cambiarse mediante la opción *Save As*. El cuerpo de la ventana está dividido en tres partes: la parte de la izquierda contiene las listas de las variables de entrada y salida globales; la parte de la derecha incluye las listas de los conjuntos de operadores, tipos de variables lingüísticas, bloques crisp y bases de reglas; por último, la zona central muestra la estructura jerárquica del sistema.

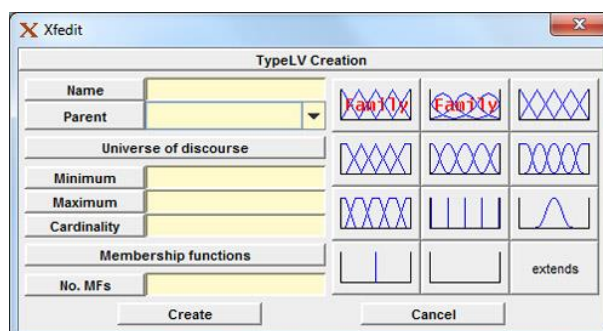
Los aceleradores para las diferentes listas son los habituales en el entorno: la tecla *Insert* crea un nuevo elemento para cada lista; la tecla *Delete* se utiliza para eliminar el elemento (cuando no ha sido usado); la tecla *Enter* o una doble pulsación permite la edición del elemento.

La creación de un sistema difuso en *Xfuzzy* usualmente comienza con la definición de conjuntos de operadores ([operator sets](#)). La figura muestra la ventana usada para editar conjuntos de operadores en *xfedit*. Tiene un comportamiento simple. El primer campo contiene el identificador del conjunto de operadores. Los restantes campos contienen listas desplegables para asignar funciones a los diferentes operadores difusos. Si la función seleccionada necesita la introducción de parámetros, se abrirá una nueva ventana para introducirlos. Las funciones disponibles en cada lista son las definidas en el paquete cargado. No es necesario seleccionar todos los campos. Una barra de comandos en la parte inferior de

la ventana presenta cuatro opciones: "Ok", "Apply", "Reload" y "Cancel". La primera opción salva el conjunto de operadores y cierra la ventana. La segunda sólo salva los últimos cambios. La tercera opción recupera los últimos valores salvados para cada campo. La última cierra la ventana desechando los cambios realizados.



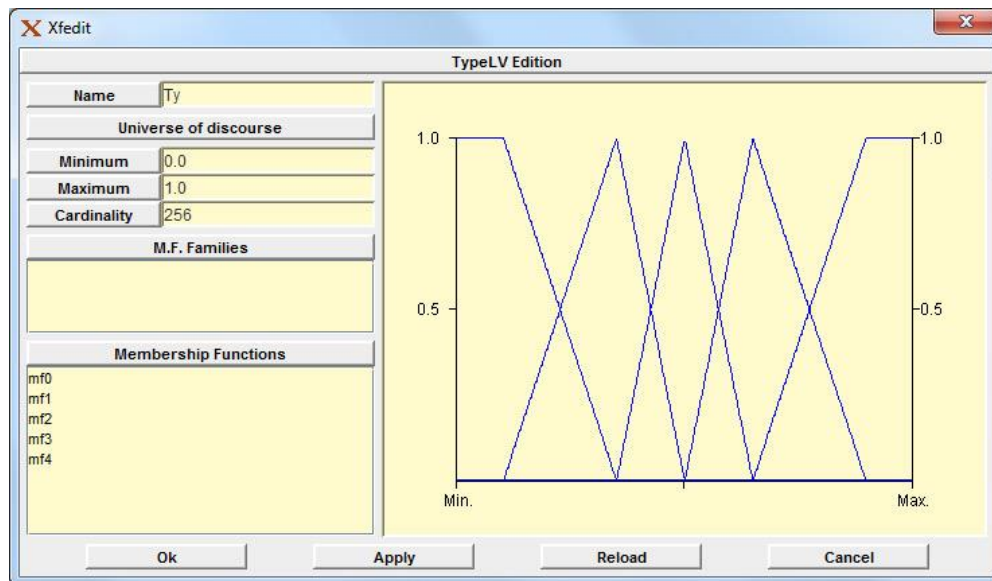
El siguiente paso en la descripción del sistema difuso es crear los tipos de las variables lingüísticas (*linguistic variable types*) mediante la ventana de *Creación de tipos* mostrada abajo. Un tipo nuevo necesita la introducción de su identificador y su universo de discurso (mínimo, máximo y cardinalidad). La ventana incluye varios tipos predefinidos correspondientes a las particiones más habituales del universo de discurso. Estos tipos predefinidos contienen distribuciones homogéneas de funciones triangulares, trapezoidales, en forma de campana y singularidades difusas. Otros tipos predefinidos son singularidades y campanas iguales que son habitualmente usadas como opción inicial para tipos de variables de salida. Cuando se selecciona uno de los tipos predefinidos es preciso introducir el número de funciones de pertenencia de la partición. Los tipos predefinidos también incluyen una opción vacía, que genera un tipo sin ninguna función de pertenencia, y la extensión de un tipo ya existente (seleccionado en el campo *Parent*), que implementa el mecanismo de herencia de XFL3.



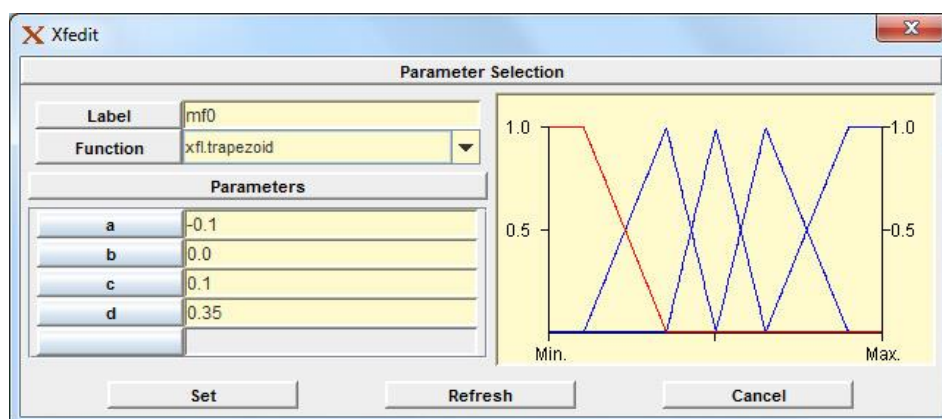
Una vez que se ha creado un tipo, éste puede ser editado usando la ventana de *Edición de tipos*. Esta ventana permite la modificación del nombre del tipo y del universo de discurso, así como añadir, editar o borrar las funciones de pertenencia del tipo editado. La ventana muestra una representación gráfica de las funciones de pertenencia donde la función seleccionada se representa con un color diferente. La parte inferior de la ventana contiene una barra de comandos con los botones habituales para salvar o descartar los cambios y para cerrar la ventana. Debemos considerar que las modificaciones en la definición del universo de discurso



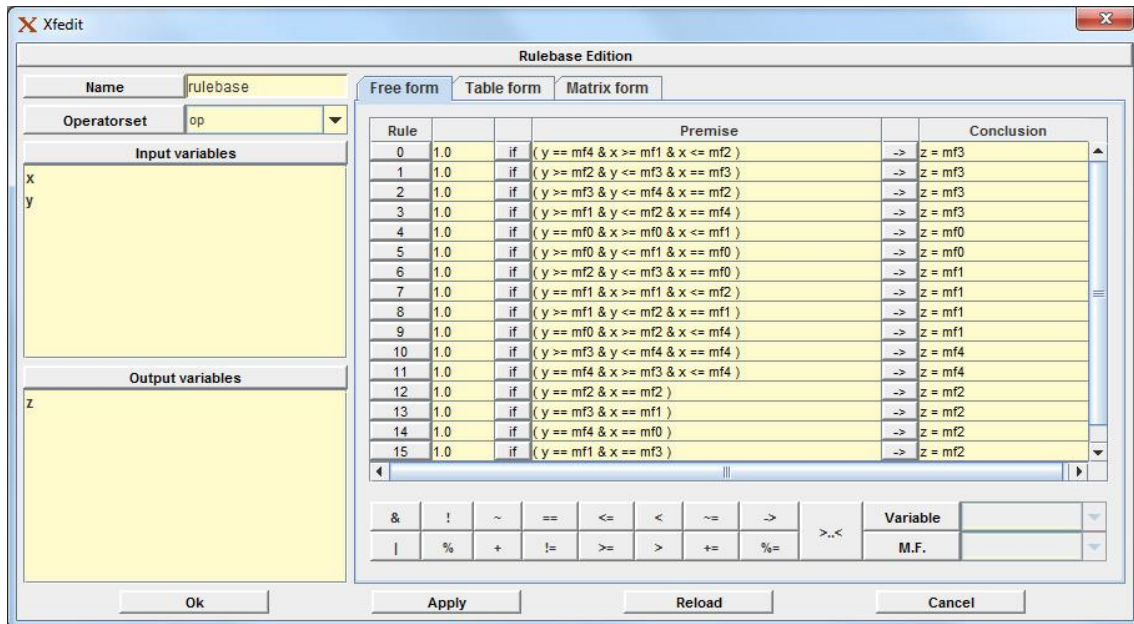
pueden afectar a las funciones de pertenencia. Por ello, se realiza una validación de los parámetros de las funciones de pertenencia antes de salvar las modificaciones, apareciendo un mensaje de error cuando la definición de una función de pertenencia se convierte en inválida.



Una función de pertenencia puede ser creada o editada a partir de la lista de funciones de pertenencia con los aceleradores habituales (tecla *Insert* y tecla *Enter* o doble pulsación). La figura anterior muestra la ventana de edición de funciones de pertenencia. La ventana dispone de campos para introducir el nombre de la etiqueta lingüística, para seleccionar la clase de función de pertenencia y para incluir los valores de los parámetros. La parte de la derecha de la ventana muestra una representación gráfica de todas las funciones de pertenencia donde la función que está siendo editada aparece con un color diferente. La parte inferior de la ventana muestra una barra de comandos con tres opciones: *Set*, para cerrar la ventana y salvar los cambios; *Refresh*, para redibujar la representación gráfica; y *Cancel*, para cerrar la ventana sin salvar las modificaciones.



El tercer paso en la definición de un sistema difuso consiste en describir las bases de reglas que expresan las relaciones entre las variables del sistema. Las [bases de reglas](#) pueden ser creadas, editadas y eliminadas de la lista correspondiente mediante los aceleradores habituales (*Insert*, *Enter* o doble click y *Delete*). La siguiente ventana facilita la edición de bases de reglas.

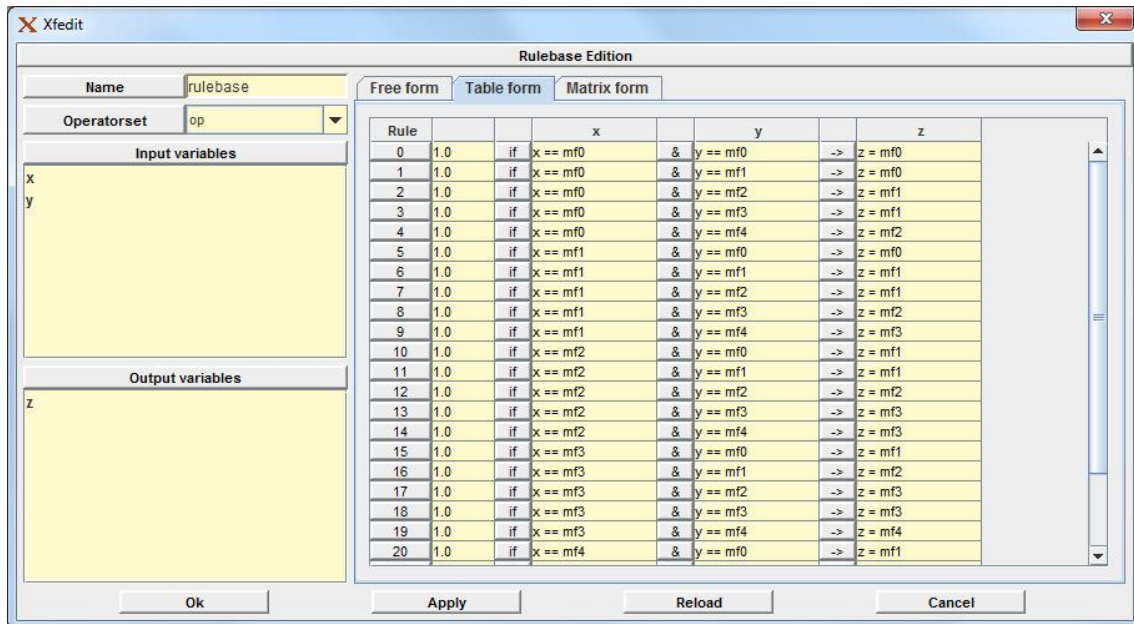


La ventana de edición de bases de reglas está dividida en tres zonas: la parte de la izquierda contiene los campos para introducir los nombres de la base de reglas y del conjunto de operadores usado, y para introducir la lista de variables de entrada y salida. La zona de la derecha se utiliza para mostrar el contenido de las reglas incluidas en la base de reglas. La parte inferior de la ventana contiene la barra de comandos con los botones habituales para salvar o descartar las modificaciones y para cerrar la ventana.

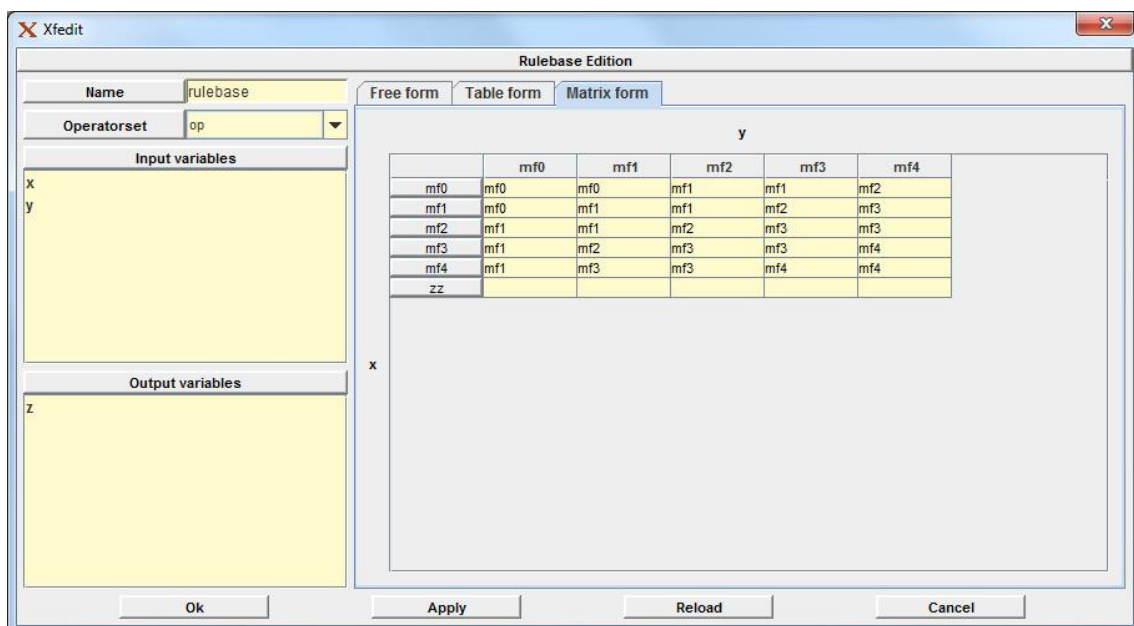
Las variables de entrada y salida pueden ser creadas, editadas o eliminadas con las secuencias de teclas habituales. La información necesaria para definir una variable es el nombre y el tipo de la variable.

Los contenidos de las reglas pueden mostrarse en tres formatos: libre, tabular y matricial. El formato libre usa tres campos por cada regla. El primero contiene el peso o factor de confianza de la regla. El segundo campo muestra el antecedente de la regla. Se trata de un campo auto-editable, en el que los cambios se llevan a cabo seleccionando el término a modificar (el símbolo "?" indica un término vacío) y usando los botones de la ventana. El tercer campo de cada regla contiene la descripción del consecuente. Es también un campo auto-editable que puede ser modificado pulsando el botón "->". Es posible generar reglas nuevas introduciendo valores en la última fila (marcada con el símbolo "\*").

La barra de botones localizada en la parte inferior de la representación de la base de reglas en formato libre permite crear términos unidos por conjunciones (botón "&") y disyunciones (botón "|"), términos modificados por los modificadores lingüísticos *not* (botón "!"), *more or less* (botón "~"), *slightly* (botón "%") y *strongly* (botón "+"), y términos simples que relacionan una variable y una etiqueta con las cláusulas *equal to* ("="), *not equal to* ("!="), *greater than* (">"), *smaller than* ("<"), *greater or equal to* (">="), *smaller or equal to* ("<="), *approximately equal to* ("~="), *strongly equal to* ("+=") y *slightly equal to* ("%="). El botón "->" se utiliza para añadir la conclusión de una regla. El botón ">.<" se emplea para eliminar un término conjuntivo o disyuntivo (p.e. el término "v == l & ?" se transforma en "v == l"). El formato libre permite describir relaciones más complejas entre las variables que los otros formatos.

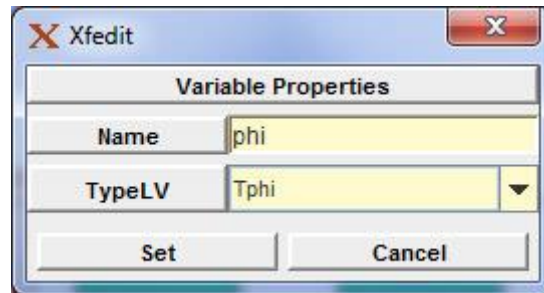


El formato tabular resulta útil para definir reglas cuyos antecedentes usan sólo los operadores *and* y *equal*. Cada regla dispone de un campo para introducir el factor de confianza y una lista desplegable por cada variable de entrada y de salida. No es necesario seleccionar todos los campos de variables, pero al menos una variable de entrada y otra de salida deben ser seleccionadas siempre. Si una base de reglas contiene una regla que no puede ser expresada en formato tabular, la tabla no puede ser abierta y se genera un mensaje de error.

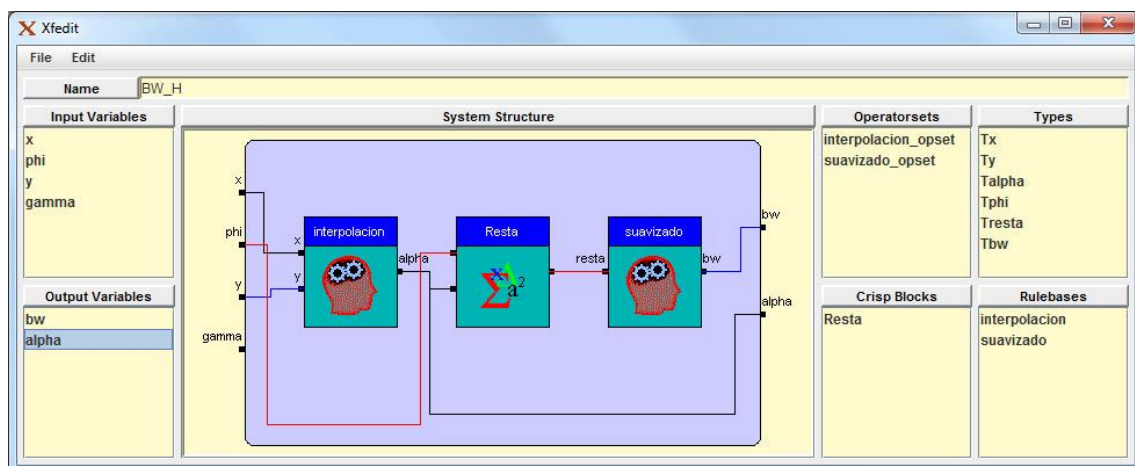


El formato matricial está diseñado específicamente para describir bases de reglas con dos entradas y una salida. Esta opción muestra el contenido de la base de reglas en un formato claro y compacto. El formato matricial genera reglas como "*if(x==X & y==Y) -> z=Z*", es decir, reglas con factor de confianza 1.0 y formadas por la conjunción de dos igualdades. Aquellas bases de reglas que no tienen el número de variables adecuado o que contienen reglas con un formato diferente no pueden ser mostradas en formato matricial.

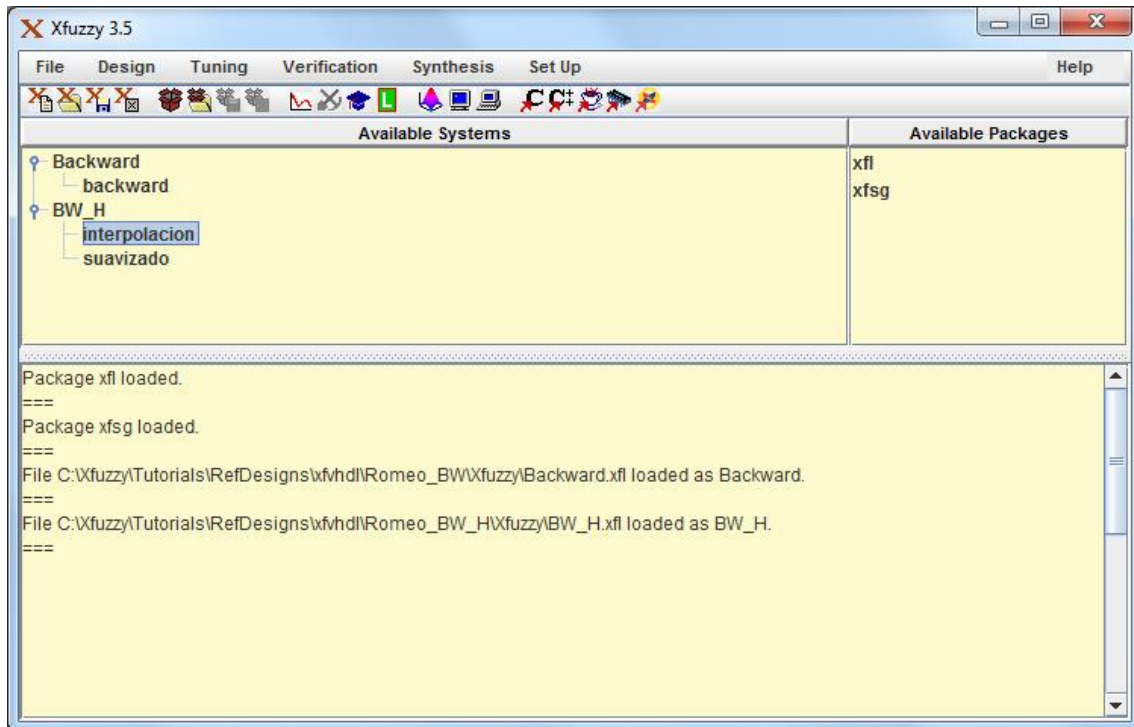
Una vez que los conjuntos de operadores, los tipos de variables y las bases de reglas han sido definidos, el siguiente paso en la definición de un sistema difuso es definir las variables de entrada y salida globales utilizando la ventana de *Propiedades de las variables*. La información necesaria para crear una variable es el nombre y el tipo de la variable.



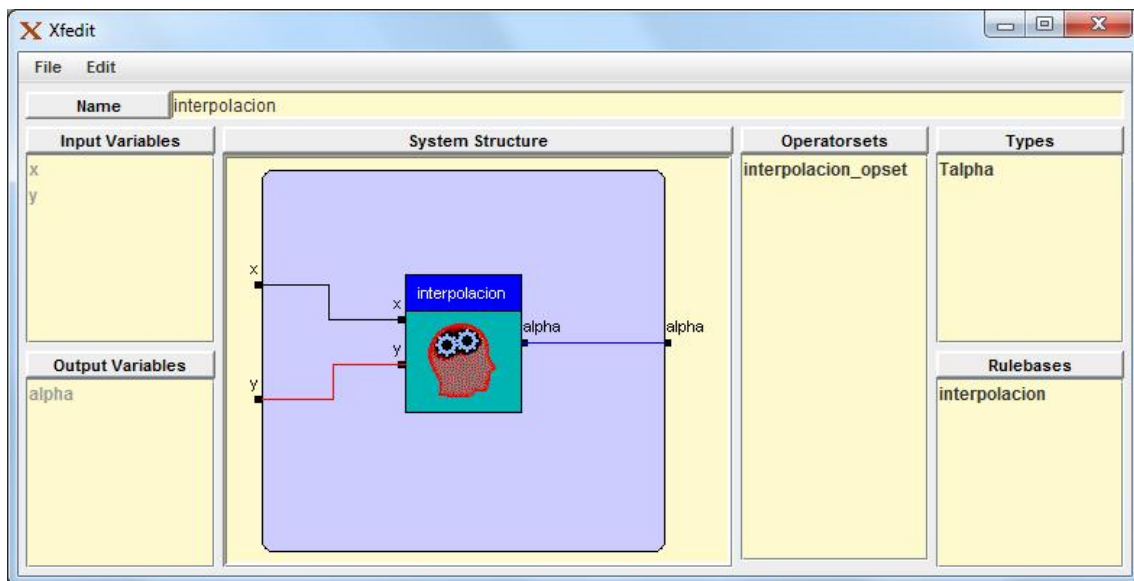
El paso final en la definición de un sistema difuso es la descripción de su estructura (posiblemente jerárquica). La tecla utilizada para introducir un nuevo módulo (una llamada a una base de reglas) en una jerarquía es la tecla *Insert*. Para establecer enlaces entre módulos, el usuario debe presionar el botón izquierdo del ratón situando el puntero sobre el nodo que representa a la variable de origen y soltar el botón con el puntero situado sobre el nodo de la variable de destino. Para eliminar un enlace, el usuario debe seleccionarlo pulsando sobre el nodo de la variable de destino y presionar la tecla *Delete*. La herramienta no permite crear lazos entre módulos.



La herramienta permite la edición individualizada de las bases de reglas de un sistema jerárquico. Para ello es preciso desplegar la jerarquía del sistema en la ventana principal de *Xfuzzy* y pulsar dos veces sobre la base de reglas que se desea editar o seleccionarla y pulsar la tecla *Insert*. Al seleccionar una base de reglas, en el menú principal se observará que algunas de las herramientas de *Xfuzzy* aparecen deshabilitadas. Ello se debe a que el uso de bases de reglas de sistemas jerárquicos está limitado a tareas de edición, ajuste, representación y síntesis.



En la ventana de *xfedit* se pueden añadir nuevos conjuntos de operadores, cambiar los tipos de las variables de salida de la base de reglas y modificar las reglas.



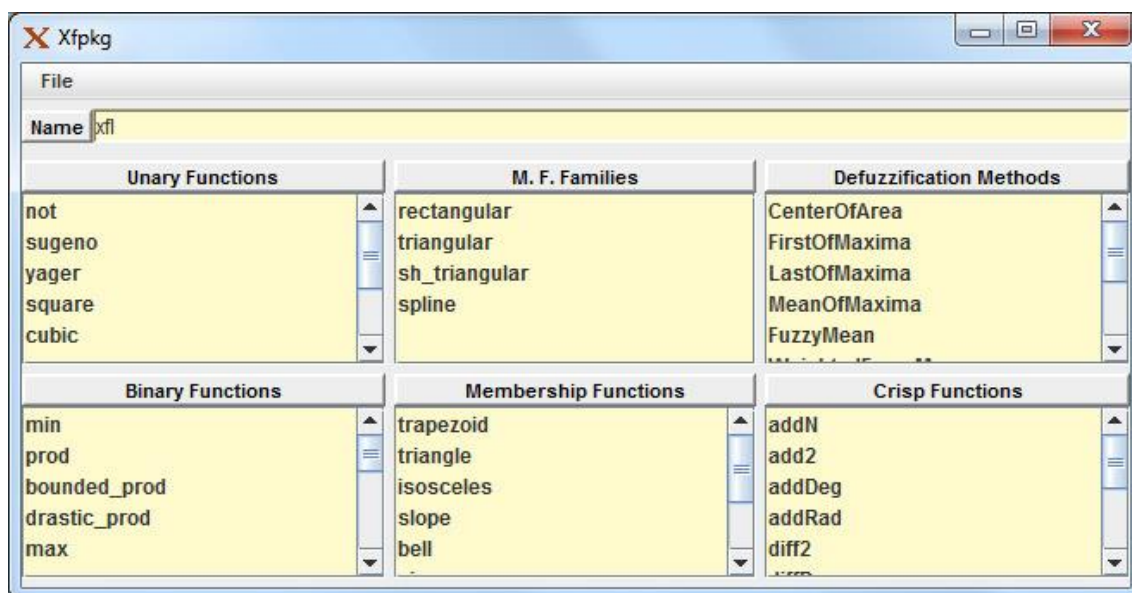
Las opciones habilitadas en la ventana de edición funcionan de forma similar a las utilizadas al editar el sistema completo. Como observación, conviene añadir que para cambiar de nombre a una base de reglas hay que acceder a la ventana de edición de base de reglas y cambiar el nombre allí.



## Herramienta de edición de paquetes – Xfpkg

La descripción de un sistema difuso en el entorno *Xfuzzy 3* se divide en dos partes. La estructura lógica del sistema (incluyendo las definiciones de conjuntos de operadores, tipos de variables, bases de reglas y estructura de comportamiento jerárquica) se especifica en ficheros con extensión ".xfl" y puede ser editada de forma gráfica con [xfedit](#). Por otra parte, la descripción matemática de las funciones usadas como conectivos difusos, modificadores lingüísticos, funciones de pertenencia, familias de funciones de pertenencia, bloques no difusos y métodos de defuzzificación se especifican en paquetes ([packages](#)).

La herramienta *xfpkg* está dedicada a facilitar la edición de paquetes. La herramienta implementa una interfaz gráfica de usuario que muestra las listas de las diferentes funciones incluidas en el paquete y los contenidos de los diferentes campos de una definición de función. La mayoría de estos campos contienen código que describe la función en diferentes lenguajes de programación. Este código debe ser introducido manualmente. La herramienta puede ser ejecutada desde la línea de comandos o desde la ventana principal del entorno, usando la opción *Edit package* en el menú *Design*.

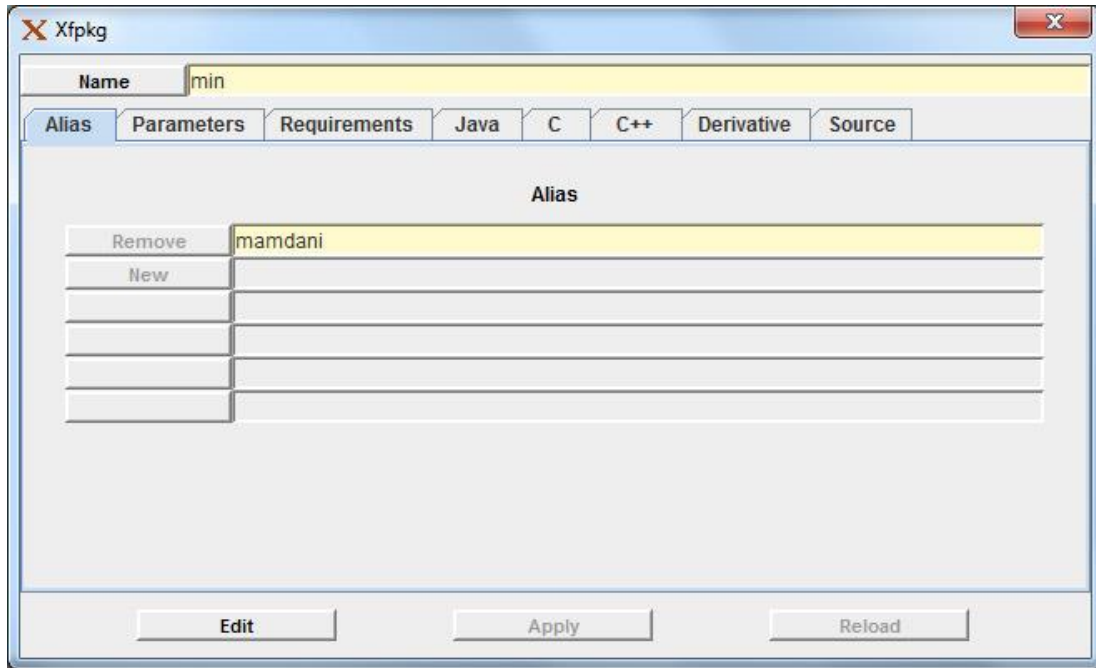


La figura anterior muestra la ventana principal de *xfpkg*. El menú *File* contiene las opciones "Save", "Save as", "Compile", "Delete" y "Close edition". Las primeras dos opciones se utilizan para salvar el paquete en un fichero. La opción "Compile" lleva a cabo el proceso de compilación que genera los ficheros ".java" y ".class" correspondientes a cada función definida en el paquete. La opción "Delete" se utiliza para eliminar el fichero que contiene el paquete y todos los ficheros ".java" y ".class" generados por el proceso de compilación. La última opción se emplea para cerrar la herramienta.

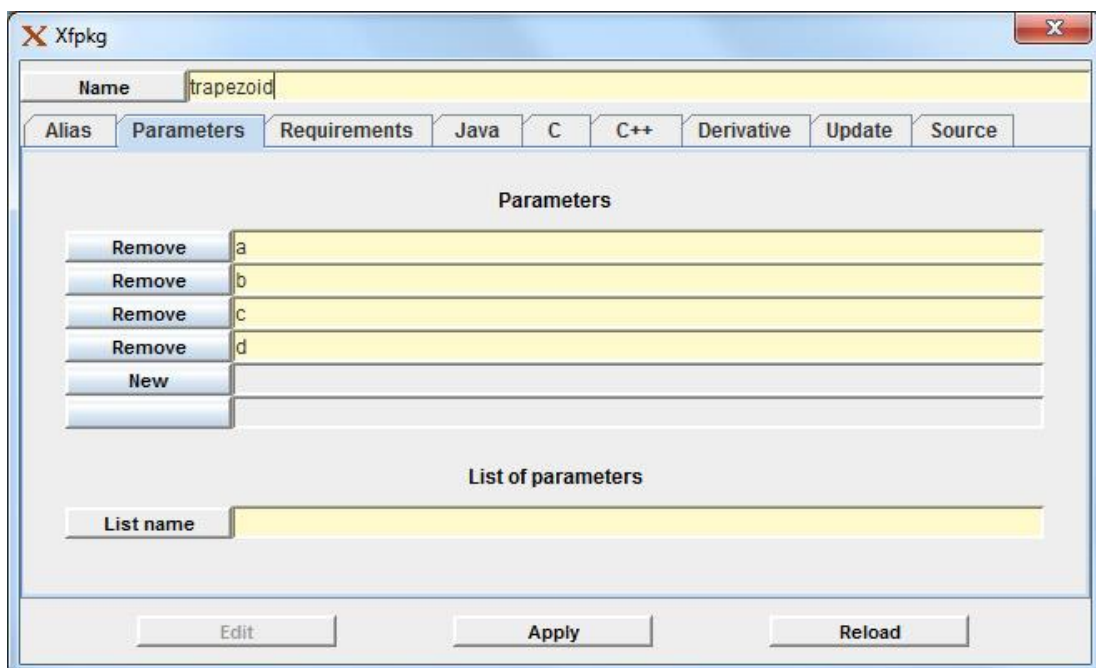
La ventana principal contiene seis listas que muestran las diferentes clases de funciones incluidas en el paquete: funciones binarias (relacionadas con los operadores de conjunción, disyunción, agregación e implicación), funciones unarias (asociadas a los modificadores lingüísticos), funciones de pertenencia (relacionadas con las etiquetas lingüísticas), familias de funciones de pertenencia (utilizadas para definir conjuntos de funciones de pertenencia relacionadas), funciones no difusas (asociadas a los bloques no difusos) y métodos de defuzzificación (usados para obtener valores representativos de las conclusiones difusas).

Pulsando sobre los elementos de estas listas se abre la ventana de edición de funciones. Esta ventana muestra el contenido de los diferentes campos de una definición de función. La parte inferior de esta zona contiene un grupo de tres botones: "Edit", "Apply" y "Reload". Al seleccionar una función de una de las listas sus campos no pueden ser editados hasta que el usuario ejecuta el comando *Edit*. El comando *Apply* salva los cambios de la definición. Esto incluye la generación de los ficheros ".java" y ".class". El comando *Reload* descarta las modificaciones realizadas y actualiza los campos con los valores previamente salvados.

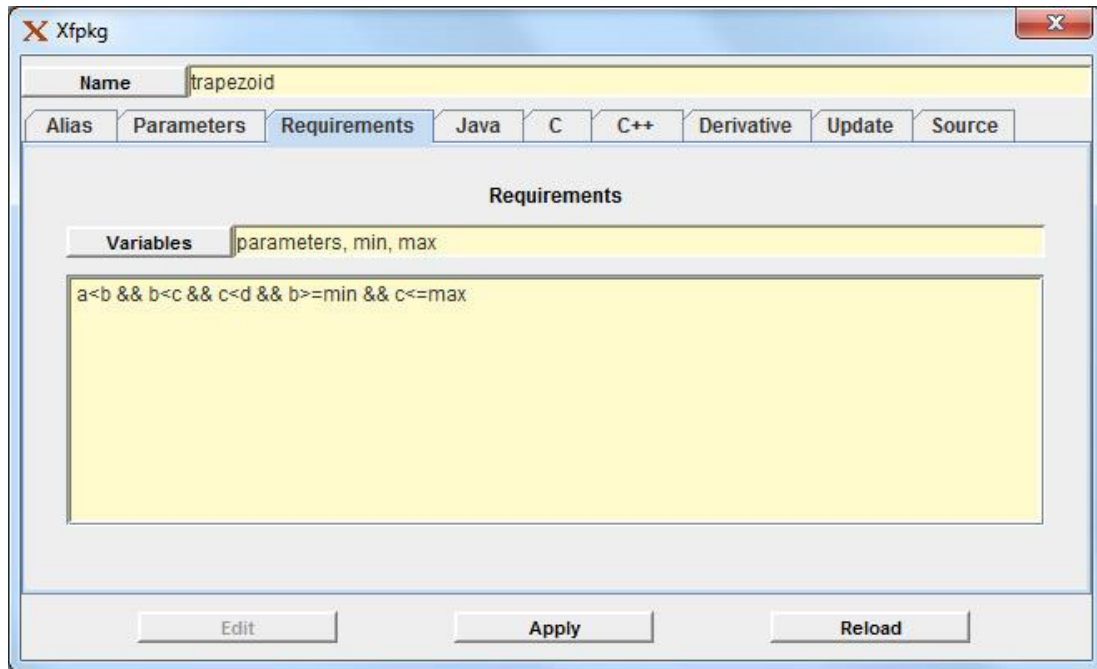
Los campos de una definición de función se distribuyen entre ocho paneles tabulados. El panel *Alias* contiene la lista de identificadores alternativos.



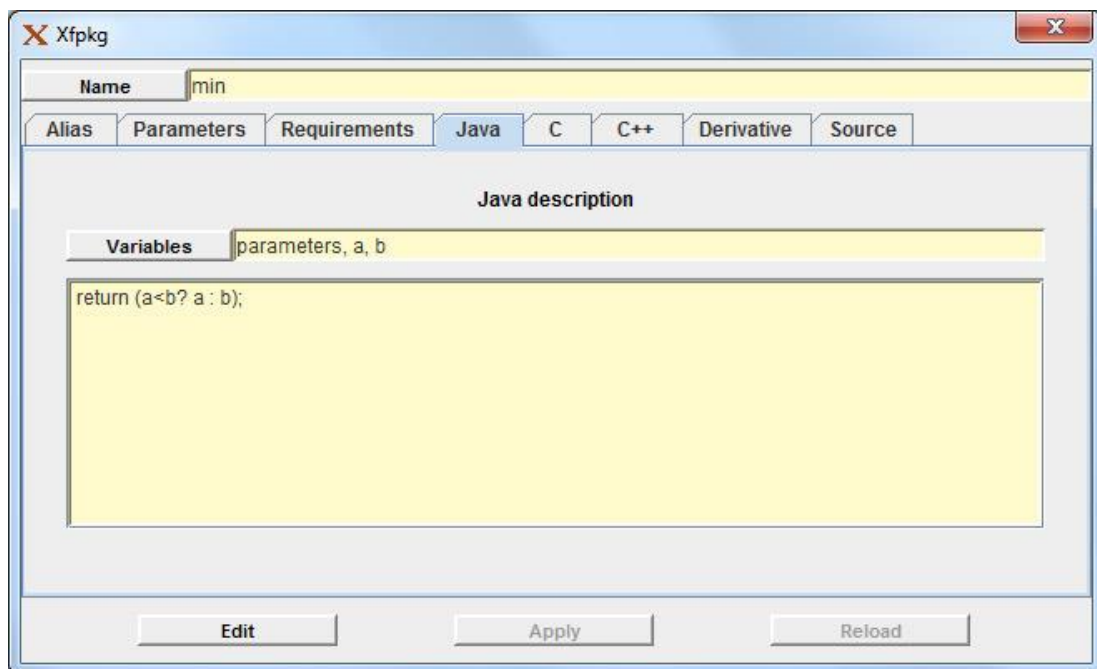
El panel *Parameters* contiene la enumeración de los parámetros usados por la función que está siendo editada.



El panel denominado *Requirements* incluye el campo donde se describen las restricciones sobre los valores de los parámetros.

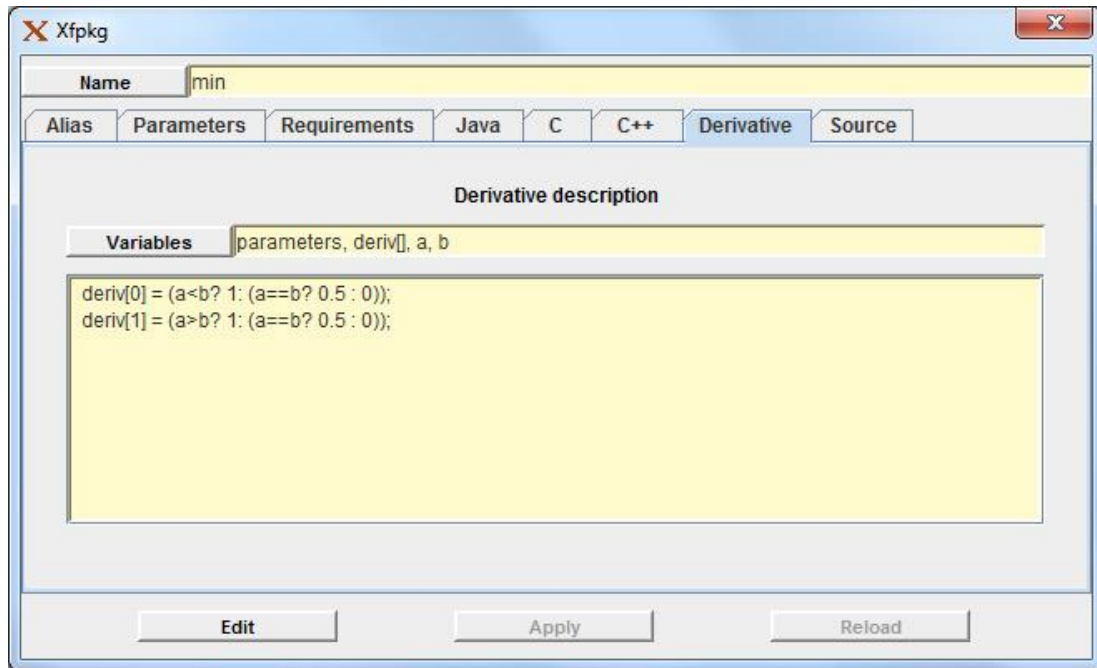


Los paneles *Java*, *C* y *C++* contienen la descripción del comportamiento de la función en estos lenguajes de programación.



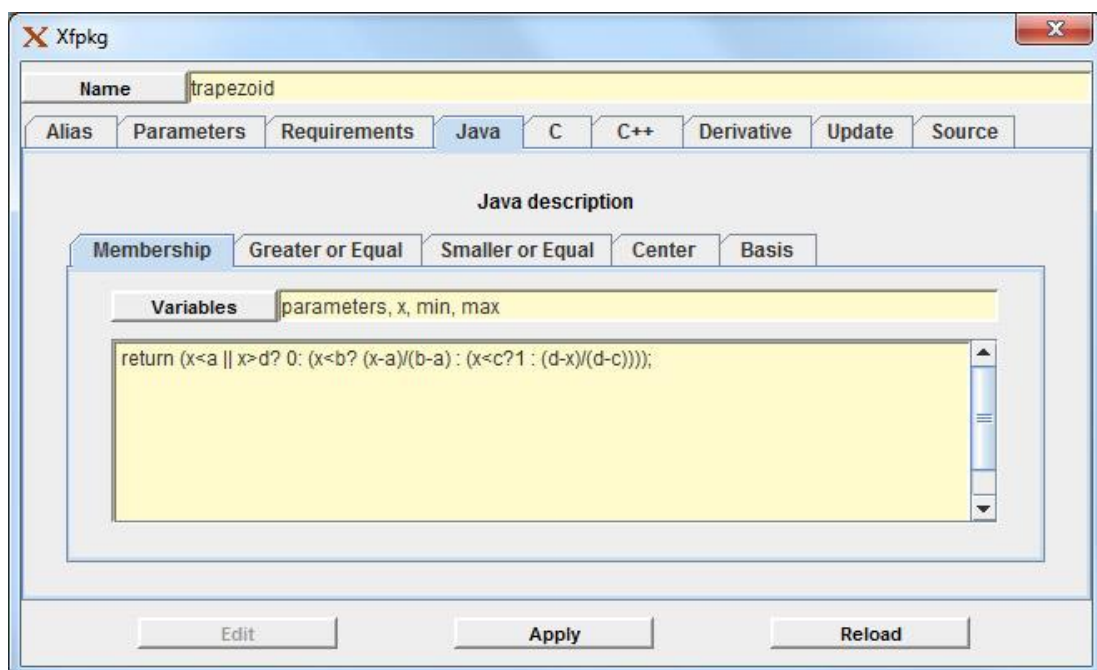
El panel *Derivative* contiene la descripción de las derivadas de la función.



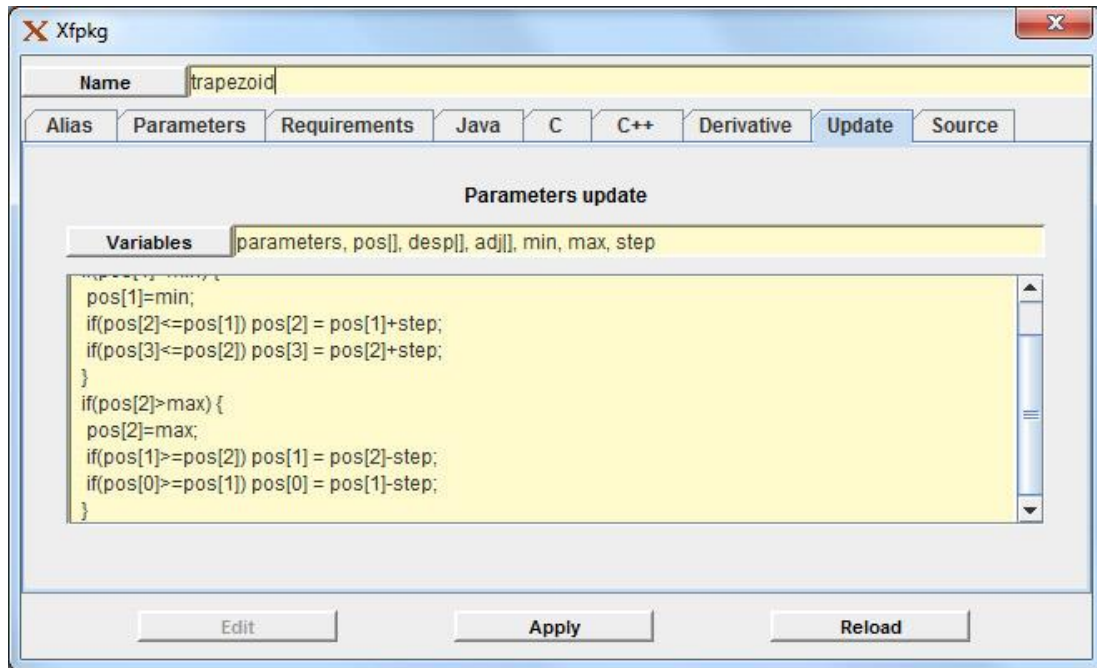


El último panel contiene los bloques fuente con el código Java de métodos locales que pueden ser usados en otros campos y que son directamente incorporados en el fichero ".java".

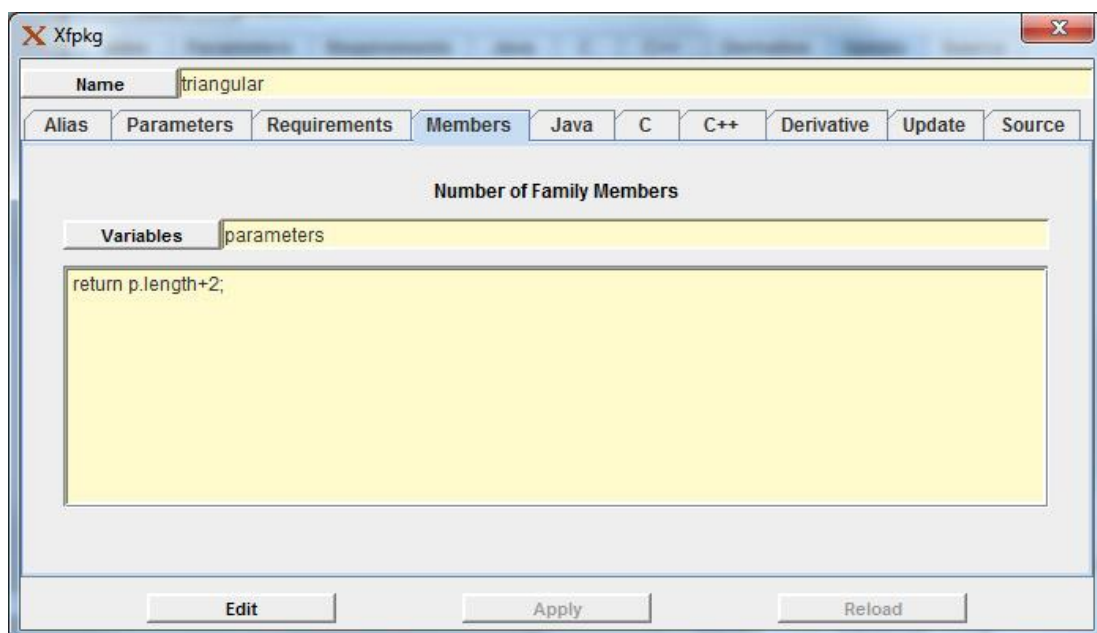
La definición de funciones de pertenencia y de familias de funciones de pertenencia necesita información adicional para describir el comportamiento de la función en los diferentes lenguajes de programación. En estos casos, los paneles *Java*, *C*, *C++* y *Derivative* contienen cinco campos para mostrar el contenido de los subbloques *equal*, *greatereq*, *smallereq*, *center*, y *basis*.



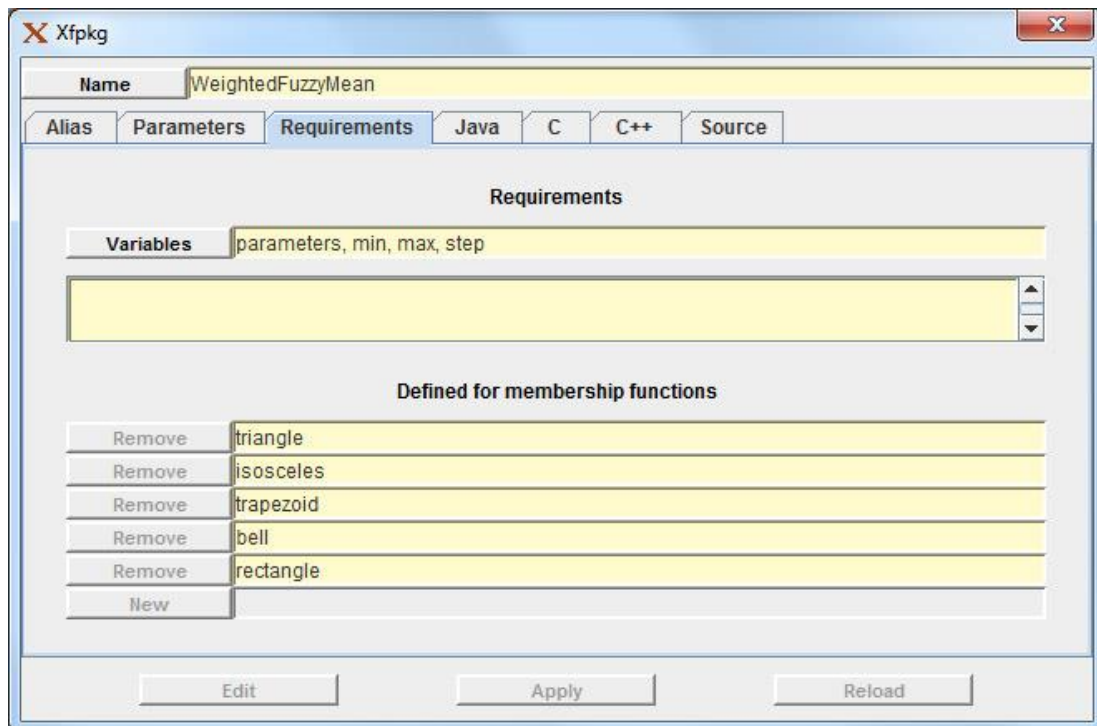
Además, las definiciones de funciones de pertenencia incluyen un panel *Update* que describe cómo modificar los valores de los parámetros de estas funciones en base a un conjunto de desplazamientos.



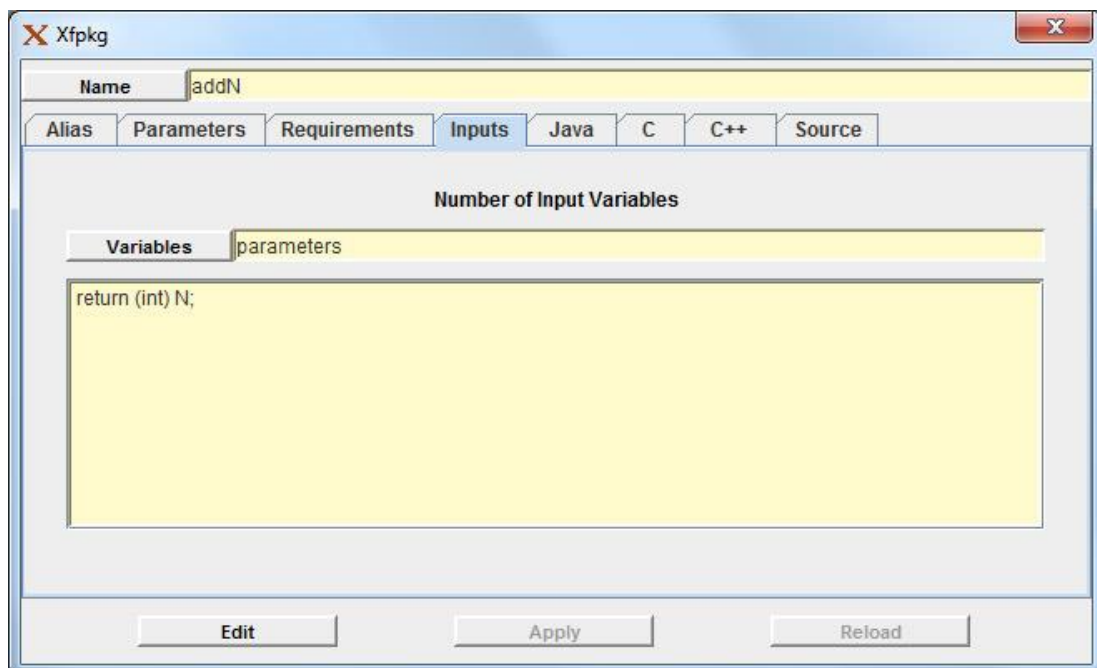
Las definiciones de familias de funciones de pertenencia contienen también un panel *Update* que describe cómo calcular el número de funciones contenidas en la familia.



Los métodos de defuzzificación pueden incluir la enumeración de las funciones de pertenencia que pueden ser usadas por cada método. Dicha enumeración aparece en el panel *Requirements*.



Finalmente, la ventana que describe a las funciones no difusas incluye también un panel llamado *Inputs* que define el número de variables de entrada de la función.



La herramienta *xfpkg* implementa una interfaz gráfica que permite al usuario visualizar y editar la definición de las funciones incluidas en un paquete. Esta herramienta se usa para describir de un modo gráfico el comportamiento matemático de las funciones definidas. En este sentido la herramienta es el complemento de *xfedit*, que describe la estructura lógica del sistema en la etapa de descripción de un sistema difuso.

## Etapa de verificación

La etapa de verificación en el proceso de diseño de sistemas difusos facilita el estudio del comportamiento del sistema difuso bajo desarrollo. El objetivo de dicho estudio es detectar las posibles desviaciones frente al comportamiento esperado e identificar las causas de estas desviaciones.

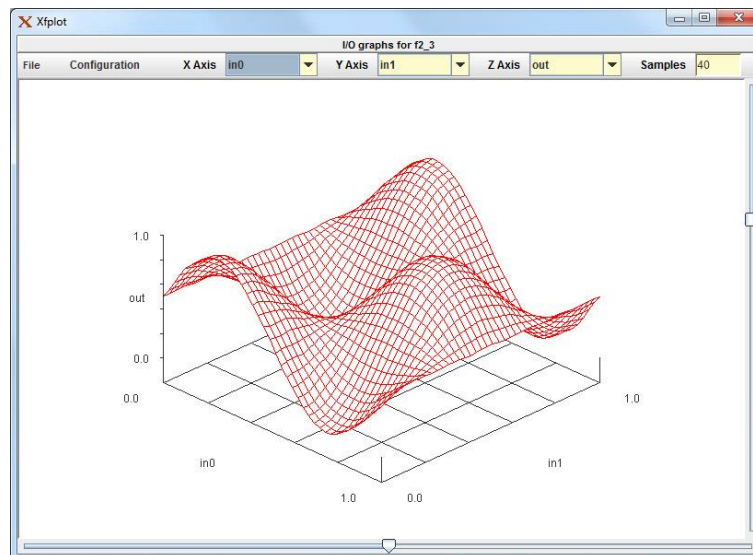
El entorno *Xfuzzy* cubre la etapa de verificación con tres herramientas. La primera de ellas es [xfplot](#), que muestra el comportamiento entrada/salida del sistema mediante una gráfica bidimensional o tridimensional. La herramienta de monitorización, [xfmt](#), muestra los grados de activación de las distintas reglas y variables lingüísticas, así como los valores de las diferentes variables internas, para un conjunto dado de entradas. Por último, la herramienta [xfsim](#) está dirigida hacia la simulación del sistema dentro de su entorno de operación (real o modelado), permitiendo ilustrar la evolución del sistema mediante representaciones gráficas de las variables seleccionadas por el usuario.



### Herramienta de representación gráfica - Xfplot

La herramienta *xfplot* ilustra el comportamiento entrada/salida de un sistema difuso (o de una determinada base de reglas de un sistema jerárquico) mediante una representación gráfica bidimensional o tridimensional. La herramienta puede ser ejecutada desde la línea de comandos con la expresión "*xfplot file.xf*", o desde la ventana principal del entorno usando la opción "*Graphical representation*" del menú *Verification*.

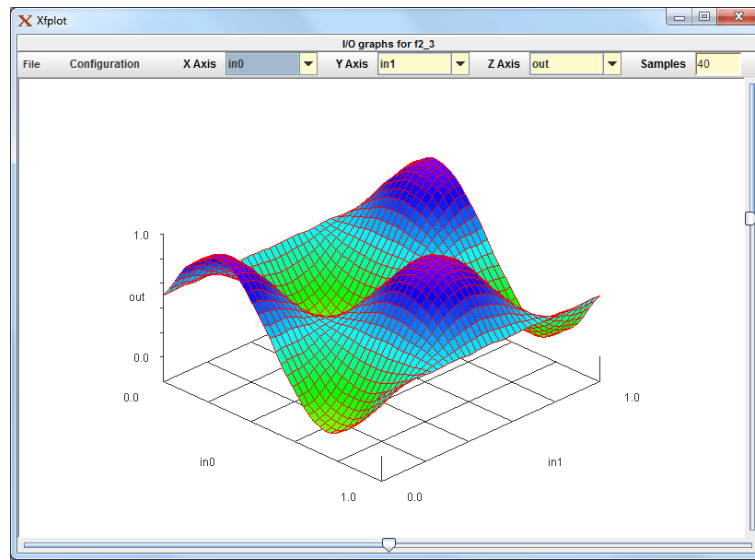
La ventana principal de la herramienta consta de un panel principal, en el que se muestra la representación gráfica, y de una barra superior, dedicada a la configuración.



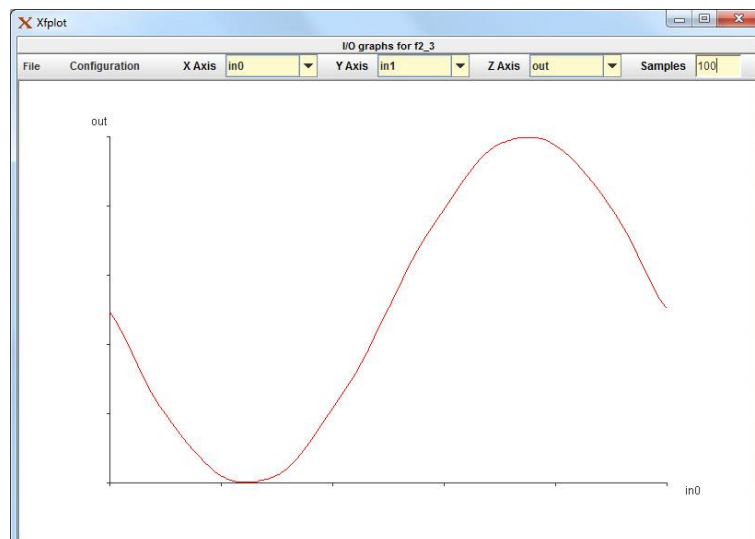
El menú *File* de la barra superior permite salvar los datos representados en un fichero externo (opción "*Save Data*"), guardar la representación gráfica como una imagen (opción "*Save image*"), actualizar la representación gráfica (opción "*Actualize*") y salir de la herramienta (opción "*Close*"). El menú *Configuration* permite seleccionar el tipo de representación (opción "*Plot Mode*"), el modelo de colores (opción "*Color Model*") y el valor de las variables de entrada no representadas (opción "*Input Values*"), así como cargar la configuración de un fichero externo (opción "*Load Configuration*") o salvarla (opción "*Save Configuration*"). Tres listas desplegables en la barra superior permiten seleccionar las variables asignadas a cada eje.

El último campo contiene el número de puntos usados en la partición de los ejes X e Y. La elección de este parámetro es importante porque determina la resolución de la representación. Un valor bajo del parámetro puede hacer que se excluyan detalles importantes del comportamiento del sistema. Por otra parte, un valor alto hará que la superficie representada sea difícil de entender al usar un grid excesivamente denso. El valor por defecto de este parámetro es 40.

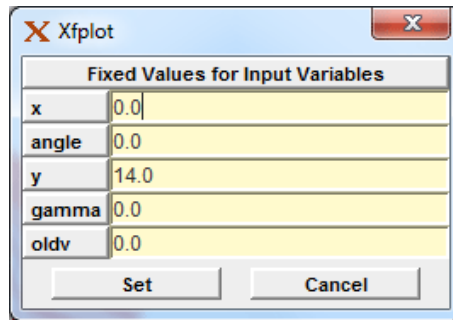
La representación gráfica tridimensional incluye la posibilidad de rotar la superficie usando los dos botones deslizantes situados en la parte derecha e inferior de la gráfica. Esta capacidad de rotación facilita la interpretación de la superficie representada.



Al seleccionar el tipo de representación bidimensional, el panel central se modifica mostrando una gráfica plana que representa la variación de la variable de salida seleccionada como eje Z con respecto a la variable de entrada seleccionada como eje X.



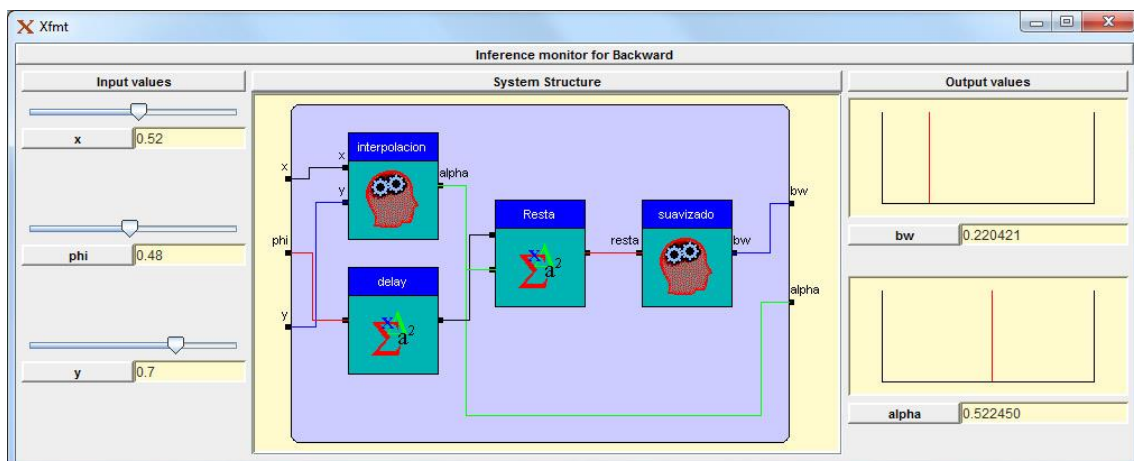
Cuando el sistema a representar contiene más variables de entrada que las requeridas por el tipo de representación seleccionado, es necesario introducir los valores asignados a las variables de entrada no representadas. Para ello se recurre a la opción "Input Values" que abre una ventana de introducción de valores.



## Herramienta de monitorización de inferencias – Xfmt

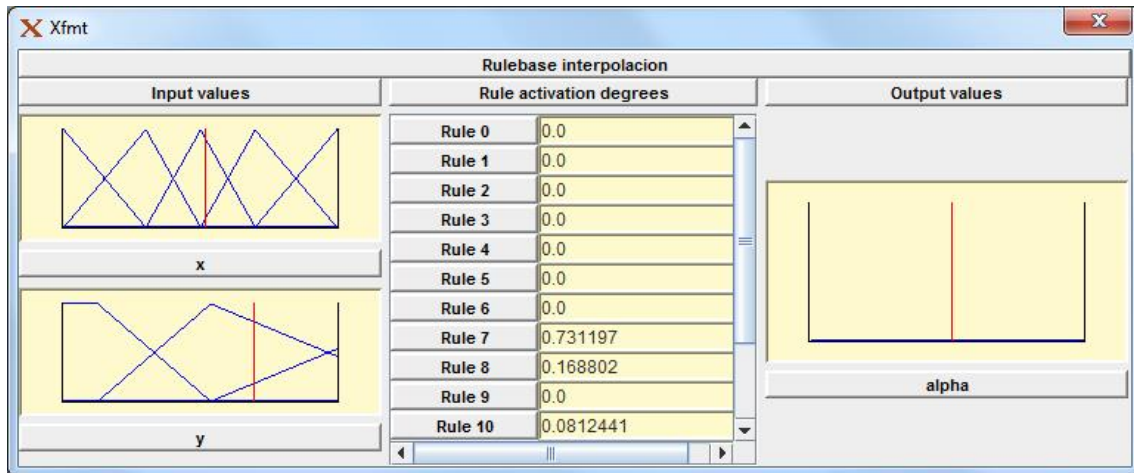
El propósito de la herramienta *xfmt* es monitorizar el proceso de inferencia del sistema, esto es, mostrar gráficamente los valores de las diferentes variables internas y los grados de activación de las reglas y las etiquetas lingüísticas para un conjunto de valores de entrada determinado. La herramienta puede ser ejecutada desde la línea de comandos con la expresión "*xfmt file.xfi*", o desde la ventana principal del entorno usando la opción "*Monitor*" del menú *Verification*.

La ventana principal de *xfmt* está dividida en tres partes. La zona de la izquierda se utiliza para introducir los valores de las variables de entrada globales. Asociado con cada variable, existe un campo para introducir manualmente el valor y un botón deslizante para introducir el valor como una posición en el rango de la variable. La parte de la derecha de la ventana muestra los conjuntos difusos asociados con los valores de las variables de salida globales, así como los valores "*crisp*" (defuzzificados) para esas variables. Estos valores son mostrados como singularidades difusas (*singletons*) en las gráficas de los conjuntos difusos (si un conjunto difuso es ya un singleton, la gráfica sólo muestra este singleton). El centro de la ventana ilustra la estructura jerárquica del sistema.



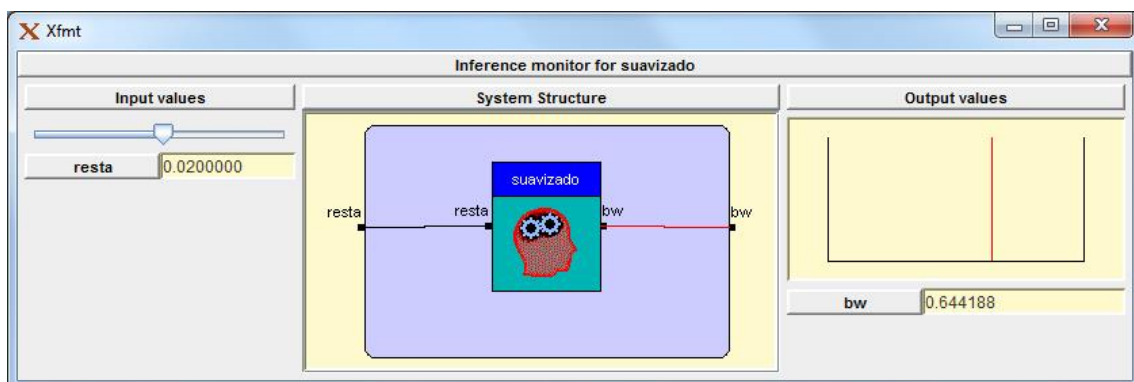
La herramienta también incluye una ventana para monitorizar los valores internos del proceso de inferencia de cada base de reglas. A esta ventana se accede pulsando sobre la base de reglas en la representación de la estructura jerárquica del sistema.





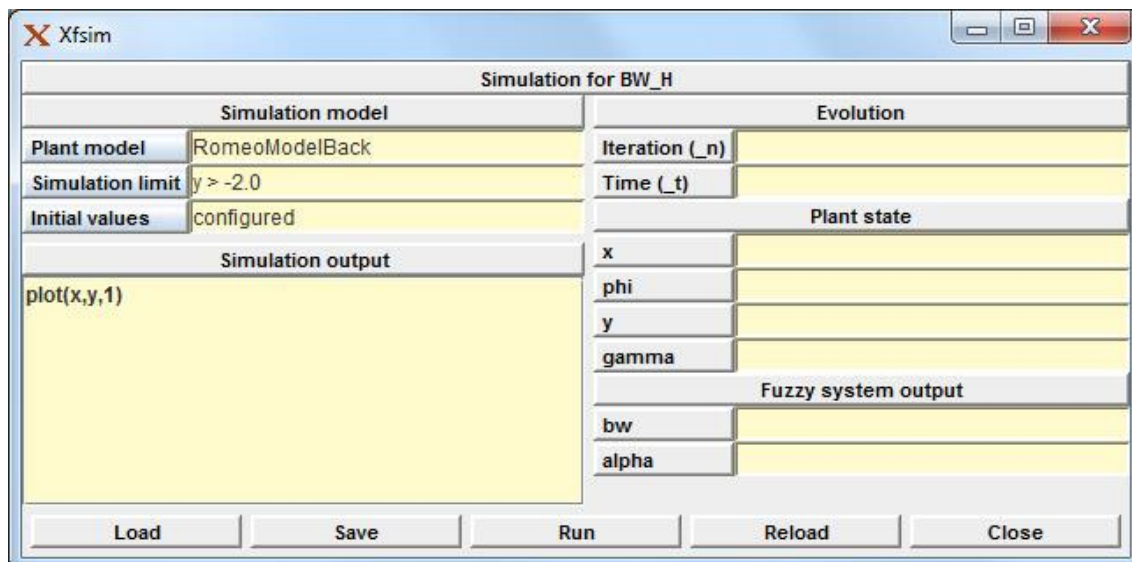
La ventana de monitorización de reglas está dividida en tres partes. Los valores de las variables de entrada se muestran en la parte izquierda como singularidades difusas sobre las funciones de pertenencia asignadas a las diferentes etiquetas lingüísticas. La parte central de la ventana contiene un conjunto de campos con los grados de activación de cada regla. En la parte de la derecha se muestran los valores de las variables de salida obtenidas en el proceso de inferencia. Si el conjunto de operadores usado en la base de reglas especifica un método de defuzzificación, el valor de salida es defuzzificado y la gráfica muestra tanto el valor difuso como el valor *crisp* finalmente asignado a la variable de salida.

Esta herramienta puede ser utilizada para monitorizar el comportamiento de cada una de las bases de reglas de un sistema de inferencia jerárquico (seleccionando cada base de reglas antes de invocar a *xfmt*). De esta manera es posible analizar el comportamiento entrada/salida de una base de reglas determinada modificando valores de variables internas del sistema.



## Herramienta de simulación – Xfsim

La herramienta *xfsim* está dirigida a estudiar sistemas realimentados. Para ello la herramienta realiza la simulación del comportamiento del sistema difuso conectado a una planta o proceso externo. La herramienta puede ser ejecutada desde la línea de comandos mediante la expresión "*xfsim file.xfl*", o desde la ventana principal del entorno con la opción "*Simulation*" del menú *Verification*.



La ventana principal de *xfsim* se muestra en la figura. La configuración del proceso de simulación se realiza en la parte izquierda de la ventana, mientras que la parte de la derecha muestra el estado del sistema realimentado. La parte inferior de la ventana contiene una barra de botones con las opciones "*Load*", "*Save*", "*Run/Stop*", "*Reload*" y "*Close*". La primera opción se utiliza para cargar una configuración para el proceso de simulación. La segunda salva la configuración actual en un fichero externo. La opción *Run/Stop* permite iniciar y parar el proceso de simulación. La opción *Reload* descarta la configuración actual y reinicia la herramienta. La última opción se emplea para salir de la herramienta.

La configuración del proceso de simulación se realiza seleccionando el modelo de la planta conectada al sistema difuso y sus valores iniciales, las condiciones de fin de simulación y la lista de las salidas que se desean obtener del proceso de simulación. Estas salidas pueden consistir en ficheros de log, para almacenar los valores de las variables seleccionadas, y representaciones gráficas de estas variables. La descripción del estado de la simulación contiene el número de iteraciones, el tiempo transcurrido desde el inicio de la simulación, los valores de las variables de entrada del sistema difuso (que representan el estado de la planta) y los valores de las variables de salida del sistema difuso (que representan la acción del sistema difuso sobre la planta).

La planta conectada al sistema difuso es descrita mediante un fichero con extensión '.class' que debe contener el código binario Java de una clase que describa el comportamiento de la planta. Esta clase debe implementar la interfaz *xfuzzy.PlantModel* cuyo código se muestra a continuación:



```

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);
}

```

La función *init()* se utiliza para inicializar la planta con sus valores por defecto. Debe generar una excepción cuando estos valores no están definidos o no pueden ser asignados a la planta. La función *init(double[])* se emplea para fijar los valores iniciales del estado de la planta a los valores seleccionados. También debe generar una excepción cuando estos valores no puedan ser asignados a la planta. La función *state()* devuelve los valores del estado de la planta (que corresponden a las variables de entrada del sistema difuso). Por último, la función *compute(double[])* modifica el estado de la planta según los valores de las variables de salida del sistema difuso. Es responsabilidad del usuario escribir y compilar esta clase Java.

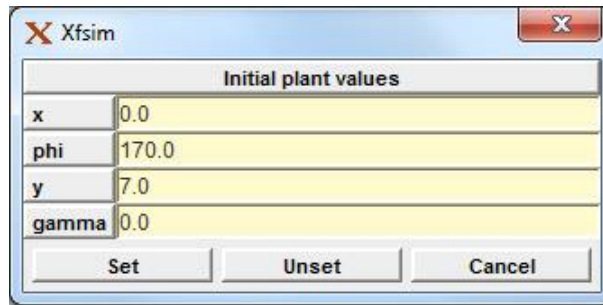
La definición de una planta mediante una clase Java proporciona una gran flexibilidad para describir sistemas externos. El modo más simple consiste en describir un modelo matemático de la evolución de la planta a partir de su estado y de los valores de salida del sistema difuso. En este esquema, las funciones *init* y *state* asignan y devuelven, respectivamente, los valores de las variables de estado internas, mientras que la función *compute* implementa el modelo matemático. Un esquema más complejo consiste en usar una planta real conectada al ordenador (usualmente mediante una tarjeta de adquisición de datos). En este caso, la función *init* debe inicializar el sistema de adquisición de datos, la función *state* debe capturar el estado actual de la planta y la función *compute* debe escribir la acción de control en la tarjeta de adquisición de datos y capturar el nuevo estado de la planta.

La configuración del proceso de simulación también requiere la introducción de alguna condición de finalización. La ventana que permite seleccionar dichas condiciones contiene un conjunto de campos con los valores límite de las variables de estado de la simulación.

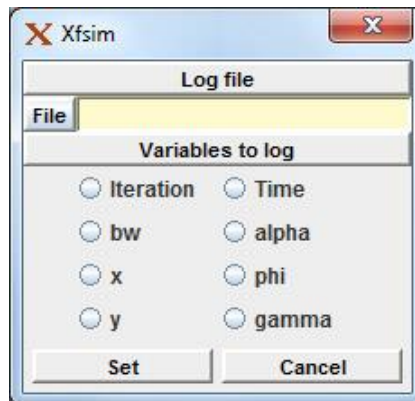
Simulation limits	
_n upper limit	1000
_t upper limit	
bw lower limit	
bw upper limit	
alpha lower limit	
alpha upper limit	
x lower limit	
x upper limit	
phi lower limit	
phi upper limit	
y lower limit	-2.0
y upper limit	
gamma lower limit	
gamma upper limit	

Buttons: Set, Unset, Cancel

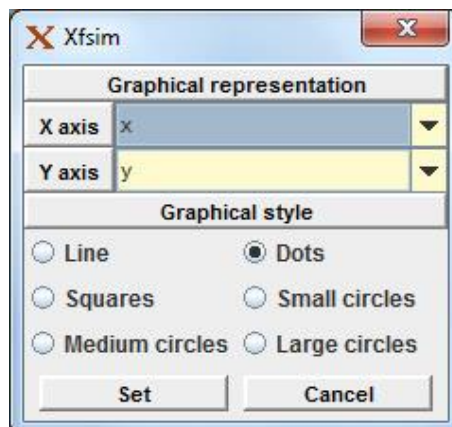
El estado inicial de la planta se describe usando la siguiente ventana que contiene el conjunto de campos relacionados con las variables de la planta.



Además de permitir almacenar los resultados de simulación en ficheros de log, la herramienta *xfsim* puede proporcionar representaciones gráficas de los procesos de simulación. Como es habitual, la tecla *Insert* se utiliza para introducir una nueva representación. Para ello se abre una ventana que pregunta por el tipo de representación: gráfica o fichero de log. La ventana para definir un fichero de log posee un campo para seleccionar el nombre del fichero y algunos botones para elegir las variables que serán almacenadas.



La ventana utilizada para definir una representación gráfica contiene dos listas desplegables para seleccionar las variables asignadas a los ejes X e Y, así como una serie de botones para elegir el estilo de representación.



La configuración del proceso de simulación puede ser salvada en un fichero externo y cargada desde un fichero previamente almacenado. El contenido de este fichero se compone de las siguientes directivas:

```

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

```

La directiva *xfsim\_plant* contiene el nombre del fichero que almacena el código binario Java que describe la planta. La directiva *xfsim\_init* contiene los valores del estado inicial de la planta. Si esta directiva no aparece en el fichero de configuración se asume para el estado inicial los valores por defecto. La directiva *xfsim\_limit* contiene la definición de las condiciones de fin de simulación, expresadas como un conjunto de límites separados por el carácter &. El formato de cada límite es "variable < valor" para los límites superiores y "variable > valor" para los inferiores. Los ficheros de log se describen mediante la directiva *xfsim\_log*, que incluye el nombre del fichero de log y la lista de las variables que van a ser almacenadas. Las representaciones gráficas se definen mediante la directiva *xfsim\_plot*, que incluye los nombres de las variables asignadas a los ejes X e Y y el estilo de representación. Un cero como valor de estilo significa gráfica con líneas; el valor 1 indica gráfica con puntos; el valor 2 hace que la gráfica utilice cuadrados; los valores 3, 4 y 5 indican el uso de círculos de diferentes tamaños.

La siguiente figura muestra un ejemplo de una clase Java que implementa el modelo de planta de un vehículo. Este modelo puede conectarse al sistema difuso *Backward* incluido entre los ejemplos de *Xfuzzy*. El estado del vehículo es almacenado en la variable interna *state[]*. Las funciones *init* asignan los valores iniciales a las componentes del estado: la primera componente es la posición X; la segunda es la orientación del vehículo frente a una dirección de referencia (*phi*); la tercera es la posición Y; la última contiene el valor actual del ángulo de giro de las ruedas (*gamma*). Estas componentes corresponden a las variables de entrada del sistema difuso, aunque la última no es utilizada por el motor de inferencia. La función *state* devuelve el valor de las variables internas. La dinámica del vehículo se describe mediante la función *compute*. Las entradas a esta función son las variables de salida del sistema difuso. *val[0]* contiene el valor objetivo de la variable *gamma* (*gref*), mientras que *val[1]* contiene el valor de la variable de salida de la primera base de reglas del sistema (*alpha*), que no es utilizado en el modelo. El cambio en el ángulo de giro del vehículo no se produce de forma instantánea, sino que presenta una cierta inercia caracterizada por una constante de tiempo definida en el modelo. En cada iteración, el nuevo valor de la variable *gamma* provoca un cambio en el ángulo de orientación y la posición del vehículo.

```

import xfuzzy.PlantModel;

public class RomeoModelBack implements PlantModel {
    private double x;
    private double y;
    private double phi;
    private double gamma;

    public RomeoModelBack() {
    }

    public void init() {
        x = 0;
        phi = 0;
        y = 0;
        gamma = 0;
    }
}

```

```

public void init(double val[]) {
    x = val[0];
    phi = val[1]*Math.PI/180;
    y = val[2];
    gamma = val[3];
}

public double[] state() {
    double state[] = new double[4];
    state[0] = x;
    state[1] = phi*180/Math.PI;
    state[2] = y;
    state[3] = gamma;
    return state;
}

public double[] compute(double val[]) {
    double LAPSE = 0.1;
    double P_TAU = 0.5;
    double v = -1.0;
    double t = 0.0;
    double gref = 1.0*val[0];
    double oldgamma = gamma;

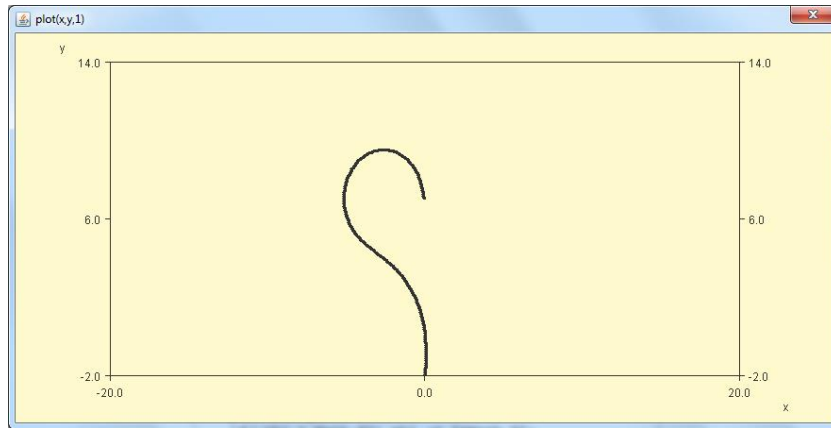
    for(t=0.0; t <= LAPSE; t+=0.001) {
        x += v*Math.sin(phi)*0.001;
        y += v*Math.cos(phi)*0.001;
        phi += v*gamma*0.001;
        if( phi > Math.PI) phi -= 2*Math.PI;
        if( phi < -Math.PI) phi += 2*Math.PI;
        gamma = gref + (oldgamma-gref)*Math.exp(-t/P_TAU);
        if( gamma > 0.4) gamma = 0.4;
        if( gamma < -0.4) gamma = -0.4;
    }
    return state();
}
}

```

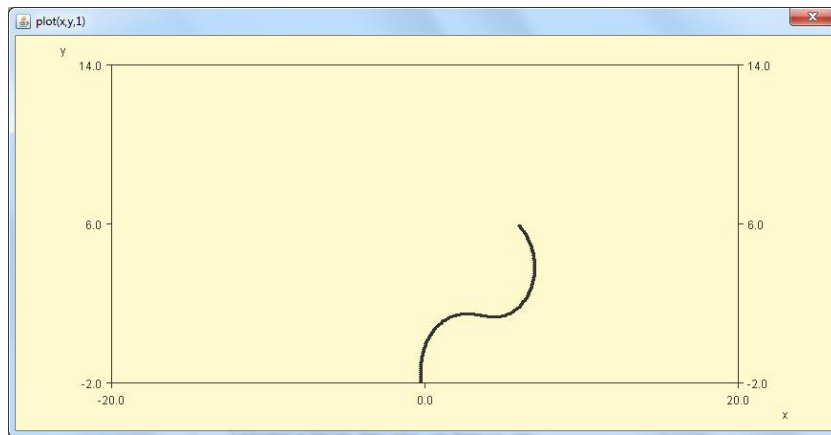
Una vez descrito el modelo de la planta, el usuario debe compilarlo para generar el fichero binario .class. Es necesario tener en cuenta que la variable de entorno CLASSPATH debe contener el camino (*path*) de la definición de la interfaz. Por tanto CLASSPATH debe incluir la ruta "*base/xfuzzy.jar*", donde *base* hace referencia al directorio de instalación de *Xfuzzy*. (Nota: en MS-Windows el path debe incluir la ruta "*base\xfuzzy.jar*").

Las siguientes figuras muestran las trayectorias seguidas por el vehículo cuando inicia el aparcamiento con diferentes condiciones iniciales.

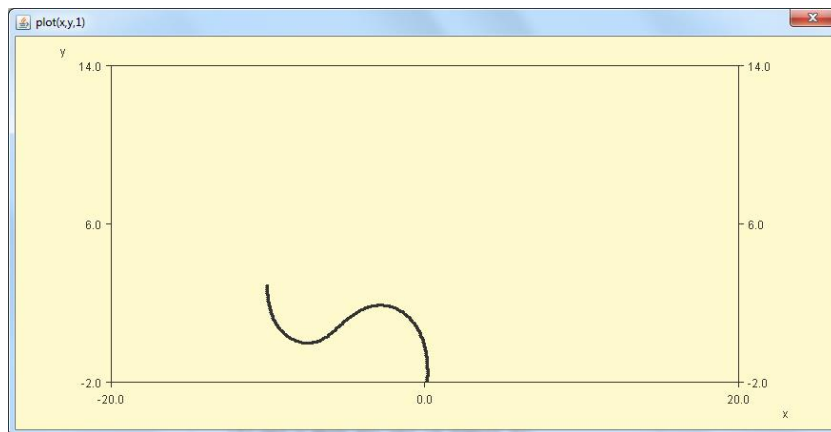
$$x = 0, y = 7, \text{ phi} = 170$$



$$x = 6, y = 6, \text{ phi} = -45$$



$$x = -10, y = 3, \text{ phi} = 0$$



## Etapa de ajuste

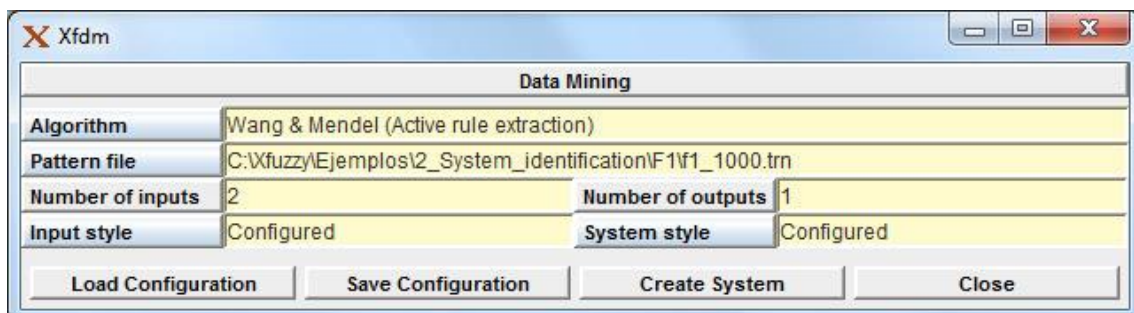
Los procesos de identificación y ajuste constituyen habitualmente las tareas más complejas en el diseño de sistemas difusos. El comportamiento de un sistema depende de la estructura lógica de sus bases de reglas y de las funciones de pertenencia de sus variables lingüísticas. El proceso de identificación tiene por objetivo determinar la estructura del sistema, mientras que el proceso de ajuste suele dirigirse normalmente a modificar los diferentes parámetros de las funciones de pertenencia que aparecen en la definición del sistema. Ya que el número de casos que deben ser considerados para extraer las reglas y el número de parámetros que deben ser modificados simultáneamente suele ser elevado, un proceso de identificación o ajuste manual resultaría claramente incómodo por lo que se requiere el uso de técnicas automáticas.

El entorno *Xfuzzy 3* incluye cuatro herramientas para esta etapa de diseño. *xfdm* y *xftsp* son herramientas de adquisición de conocimiento. La primera permite obtener la estructura de sistemas de inferencia utilizados como aproximadores o clasificadores difusos, mientras que la segunda está especialmente enfocada a la predicción de series temporales. *xfsl*, es una herramienta de ajuste de parámetros basada en el uso de algoritmos de aprendizaje supervisado. En las técnicas de aprendizaje supervisado el comportamiento deseado del sistema es descrito mediante un conjunto de patrones de entrenamiento (y de test). El aprendizaje supervisado intenta minimizar una función de error que evalúa la diferencia entre el comportamiento actual del sistema y el comportamiento deseado definido mediante el conjunto de patrones de entrada/salida. Por último, *xfsp* es una herramienta de simplificación que permite reducir el número de funciones de pertenencia y compactar las bases de reglas de un sistema difuso para facilitar su implementación software o hardware e incrementar su interpretabilidad lingüística.

### Herramienta de adquisición de conocimiento - Xfdm

La herramienta *xfdm* facilita la identificación de sistemas difusos a partir de datos numéricos empleando distintos algoritmos basados en técnicas de particionamiento matricial (*Grid Partitioning*) o de agrupamiento de datos (*Cluster Partitioning*). *xfdm* puede ejecutarse desde la línea de comandos, o a través de su interfaz gráfica utilizando la opción "*Data Mining*" del menú *Tuning* o el icono correspondiente en la ventana principal del entorno.

La ventana principal de *xfdm* está dividida en dos partes. La parte superior se utiliza para configurar el proceso de identificación: selección del algoritmo empleado, fichero de datos de entrada/salida, número de entradas y salidas, estilo de las entradas y estilo del sistema difuso.



Los botones situados en la parte inferior de la ventana permiten, respectivamente, cargar o salvar una configuración determinada, crear el sistema difuso y cerrar la interfaz gráfica de la herramienta.

## Algoritmos

*xfdm* incorpora distintos algoritmos de identificación agrupados en dos categorías:

### a) Algoritmos basados en estructura (*Structure-oriented algorithms*)

Estos algoritmos realizan una partición fija o variable de los universos de discurso de las variables de entrada y analizan los datos numéricos que describen el comportamiento del sistema para asignar una regla por cada línea del fichero. Posteriormente, resuelven los conflictos que puedan haberse producido y seleccionan las reglas del sistema difuso en función de su grado de activación y de los parámetros de configuración definidos por el usuario. *Xfdm* incluye tres algoritmos de identificación que trabajan con particiones fijas (*Wang & Mendel*, *Nauck* y *Sendhadji*) y uno más que contempla un número variable de particiones (*Incremental Grid*). Adicionalmente, la opción "*Flat System*" permite generar especificaciones de sistemas difusos con un comportamiento plano que pueden resultar de utilidad como entrada a la herramienta de entrenamiento o a otras facilidades de *Xfuzzy*.

Los parámetros y opciones específicas de estos algoritmos son:

- Nauck:
  - *Number of rules*: número de reglas a identificar
  - *Type of selection*: "*Best rules*" o "*Best per class*"
- Sendhadji:
  - *Number of rules*: número de reglas a identificar
- Incremental Grid:
  - *Limit of MFCs*, *Limit of Rules*, *Limit of RMSE*: la ejecución del algoritmo termina cuando se alcanza uno de estos límites.
  - *Learnig option*: opción de ajuste activada/no activada

### b) Algoritmos basados en agrupamiento (*Cluster-oriented algorithms*)

*xfdm* incorpora también otros algoritmos para generar un sistema difuso a partir de una serie de datos empleando técnicas de agrupamiento o *Clustering*. Al agrupar conjuntos de puntos en clusters representados por puntos prototipo, este tipo de técnicas permiten reducir considerablemente la información que debe manejar el algoritmo y dan lugar habitualmente a sistemas difusos con menor número de reglas. La herramienta incluye cuatro algoritmos que utilizan un número fijo de clusters (*Hard C-Means*, *Fuzzy C-Means*, *Gustafson-Kessel* y *Gath-Geva*), así como dos algoritmos que permiten variar de forma iterativa el número de clusters hasta alcanzar el límite definido por el usuario (*Incremental Clustering* e *ICFA*).

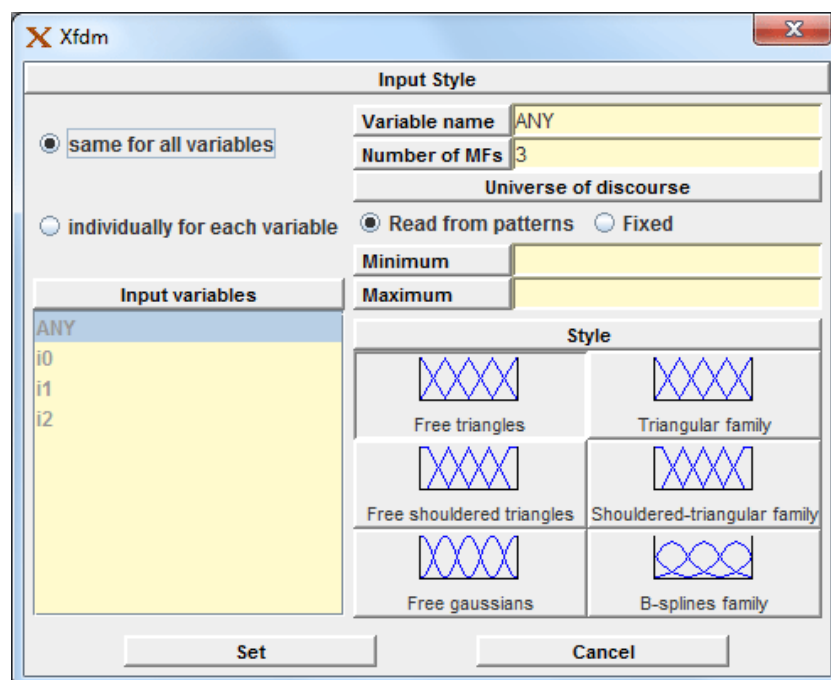
Los parámetros y opciones específicas de estos algoritmos son:

- Incremental Clustering:
  - *Neighborhood radius*: radio de vecindad
  - *Max. N. of clusters*: máximo número de clusters

- Fixed Clustering:
  - Clustering algorithm: *Hard C-Means, Fuzzy C-Means, Gustafson-Kessel, Gath-Geva*
  - Number of clusters
  - Limit on iterations
  - Fuzziness index
  - Limit on cluster variation
  - Learning option: *Activate/No activate*
  
- ICFA (Incremental Clustering for Function Approximation):
  - Number of clusters
  - Max. Iterations
  - Fuzziness index
  - Limit on cluster variation
  - Activate migration: *activada/no activada*

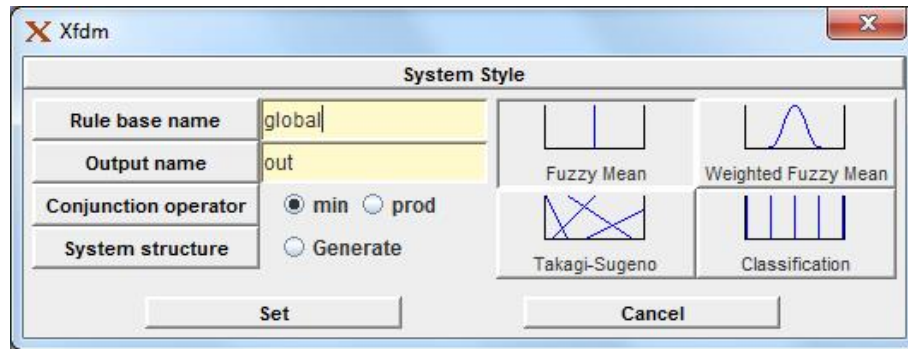
### Selección de estilos

La interfaz gráfica para la selección de estilo de las variables de entrada del sistema permite elegir, de forma conjunta para todas las variables o independientemente para cada una de ellas, el rango, el número y el tipo de funciones de pertenencia. Las opciones disponibles incluyen funciones de pertenencia (libres o agrupadas en familias) lineales a tramos, gaussianas y basadas en splines.



Por otra parte, la interfaz gráfica para la selección del estilo del sistema permite elegir el operador de conjunción usado para implementar el conectivo de antecedentes de las reglas, así como el método de defuzzificación utilizado. En este último caso, las posibles alternativas son: Fuzzy Mean, Weighted Fuzzy Mean, Takagi-Sugeno de primer orden y Max Label (para clasificadores difusos).





### Fichero de configuración

La configuración de un proceso de identificación puede ser guardada en y cargada desde un fichero externo. El contenido de este fichero está formado por las siguientes directivas:

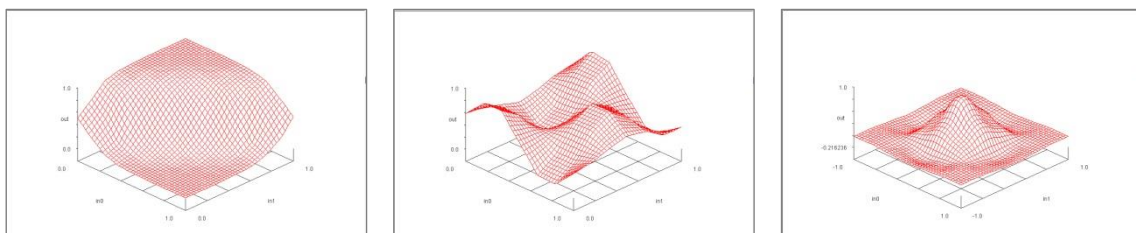
```

xfdm_pattern("file_name")
xfdm_inputs(n_inputs)
xfdm_outputs(n_outputs)
xfdm_input(variable|ANY,min,max,N_MFs,style)
xfdm_system(rulebase_name,out_name,and_op,gen,style)
xfdm_algorithm(algorithm_name,[value],...)

```

La directiva *xfdm\_pattern* selecciona el fichero de patrones de entrenamiento utilizado para identificar el sistema. *xfdm\_inputs* y *xfdm\_outputs* especifican el número de entradas y salidas, respectivamente. El estilo de las variables de entrada se define mediante una o más directivas *xfdm\_input*, cuyos parámetros indican el nombre de la variable ('ANY' para todas las del sistema), el rango de valores ('0.0, 0.0' si se obtiene del fichero de patrones), el número de funciones de pertenencia y el estilo de las mismas (0: *Free triangles*; 1: *Triangular family*; 2: *Free shouldered triangles*; 3: *Shouldered-triangular Family*; 4: *Free gaussians*; y 5: *B-splines family*). La directiva *xfdm\_system* define el estilo del sistema, incluyendo como parámetros el nombre de la base de reglas, el nombre de la variable, el operador usado como conectivo de antecedentes (0: *min*; 1: *prod*), la opción de generación del sistema (0: solo identifica la base de reglas; 1: también genera la estructura del sistema) y el método de defuzzificación empleado (0: *FuzzyMean*; 1: *WeightedFuzzyMean*, 2: *Takagi-Sugeno*; y 3: *MaxLabel*). Por último, el algoritmo de identificación, así como sus posibles parámetros, se define mediante la directiva *xfdm\_algorithm*.

La siguiente figura muestra algunos ejemplos de sistemas difusos para aproximación de funciones generados con *xfdm*.

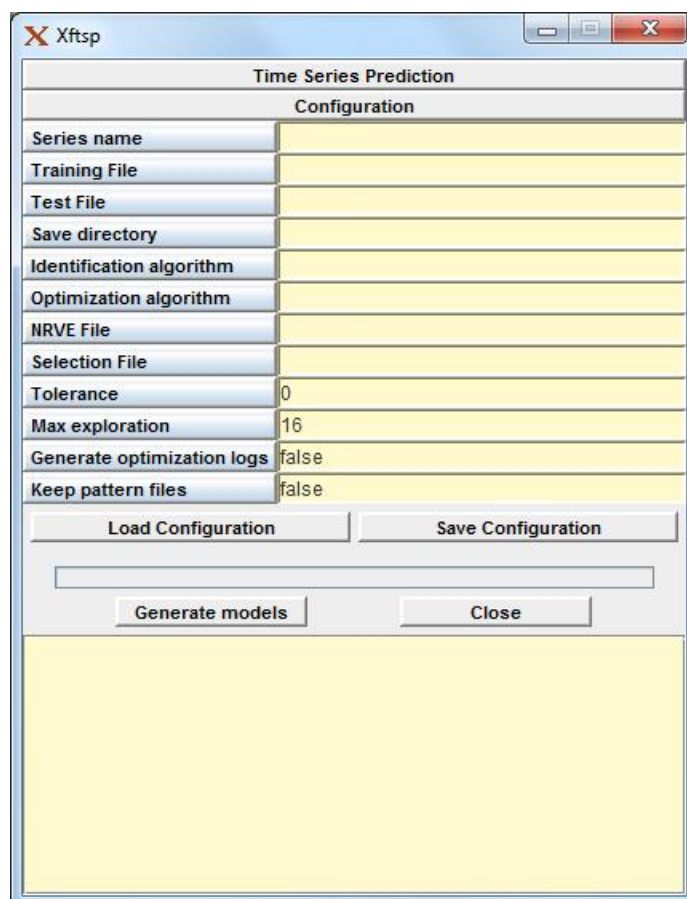


## Herramienta de predicción de series temporales - Xftsp

La herramienta *xftsp* genera sistemas de inferencia difusos que implementan modelos autoregresivos para la predicción a corto y largo plazo de series temporales. Para ello aplica una metodología basada en el uso de estimaciones no paramétricas del ruido o varianza residual (para seleccionar el número óptimo de variables de entrada) en combinación con las herramientas de identificación y aprendizaje supervisado de *Xfuzzy* (para determinar la estructura de los sistemas)<sup>1</sup>.

Esta metodología responde a una estrategia de predicción directa, lo que implica la construcción de un autorregresor por cada uno de los términos del horizonte de predicción deseado. En cada caso, el subconjunto óptimo de entradas es seleccionado a priori mediante una estimación de ruido no paramétrica (por ejemplo, el Delta Test). La especificación del sistema difuso correspondiente a cada horizonte de predicción se obtiene entonces mediante un proceso iterativo en el que se realizan sucesivas fases de identificación y ajuste, incrementando el número de etiquetas lingüísticas de las entradas, hasta que el error del sistema entra en el rango estimado previamente.

*Xftsp* se puede ejecutar en modo gráfico, utilizando la opción "Time Series Prediction" del menú *Tuning* o el icono correspondiente en la ventana principal del entorno, o desde la línea de comandos empleando un archivo de configuración.



<sup>1</sup> F. Montesino, A. Lendasse, A. Barriga  
*Autoregressive time series prediction by means of fuzzy inference systems using nonparametric residual variance estimation*  
 Fuzzy Sets and Systems 2010  
 DOI: [10.1016/j.fss.2009.10.018](https://doi.org/10.1016/j.fss.2009.10.018)

La interfaz gráfica de *xftsp* permite recopilar la información necesaria para ejecutar la herramienta. Dicha información incluye los siguientes items:

- *Series name*: Nombre de la serie temporal
- *Training file*: Fichero de patrones de entrenamiento
- *Test file*: Fichero de patrones de test
- *Save directory*: Directorio donde se almacenan los ficheros de salida
- *Identification algorithm*: Algoritmo utilizado en la fase de identificación ([xfdm](#))
- *Optimization algorithm*: Algoritmo utilizado en la fase de optimización ([xfs!](#))
- *NRVE file*: Estimación no paramétrica de la varianza residual para cada horizonte temporal
- *Selection file*: Fichero de selección de variables de entrada para cada horizonte temporal (\*)
- *Tolerance*: Permite definir la estimación empleada para determinar la complejidad del sistema difuso como un valor fijo o que se incrementa al aumentar el horizonte de predicción
- *Max exploration*: Número máximo de funciones de pertenencia por entrada
- *Generate optimization logs*: Conserva los ficheros de log generados por la ejecución de *xfs!* en la fase de optimización de todos los sistemas difusos
- *Keep pattern files*: Conserva en los directorios '*xftsp-step-\**' los ficheros de patrones de entrenamiento (y test) que se utilizan en las fases de identificación y optimización

(\*) En *Xfuzzy*, los errores suelen normalizarse frente al rango de la serie al cuadrado, por lo que las estimaciones deben normalizarse de acuerdo con esta circunstancia.

La zona central de la interfaz gráfica de *xftsp* contiene cuatro botones separados por una barra de progreso de ejecución. Los dos botones superiores permiten cargar (*Load Configuration*) o guardar (*Save configuration*) un fichero de configuración.

La sintaxis de las distintas directivas que pueden aparecer en el fichero de configuración se muestra a continuación:

```
xftsp_series_name("name")
xftsp_training_file("file_name")
xftsp_test_file("file_name")
xftsp_id_algorithm(algorithm_name, value,...)
xftsp_opt_algorithm(algorithm_name, value,...)
xftsp_nrve("file_name")
xftsp_selection("file_name")
xftsp_option(tolerance, increment)
xftsp_option(max_exploration, max_num_MFs)
xftsp_option(generate_optimization_logs)
xftsp_option(keep_pattern_files)
```

El número de filas del fichero NRVE determina el horizonte temporal que se desea predecir y, por tanto, el número de sistemas difusos que serán creados. Por otra parte, el número de columnas del fichero de selección de entradas fija el tamaño máximo de los autoregresores, esto es, el máximo número de variables de entrada de los sistemas difusos.

Una vez completada la configuración, el botón *Generate models* permite lanzar el proceso de generación de los sistemas difusos que modelan la serie temporal. La mayoría de los mensajes generados durante la ejecución de la herramienta son mostrados por la salida estándar, es decir, la ventana de comandos desde la que se lanzó *Xfuzzy* o se ejecutó el comando *xftsp*. Estos mensajes son también escritos en un fichero de log, denominado 'xftsp-run-results.log', que acumula numerosos comentarios asociados a los distintos pasos de ejecución de la herramienta. Cuando se ejecuta *xftsp* desde la interfaz gráfica, los mensajes relacionados con la carga y almacenamiento de ficheros de configuración, así como el aviso de fin de ejecución se muestran en la zona inferior de dicha interfaz. Las primeras líneas del fichero de log resultante de una ejecución de *xftsp* presentan el siguiente aspecto:

```
Date: Sat Mar 03 08:39:59 CET 2018
Series name: estsp07
Training series file: C:\workspace\Ejemplos\Tools\xftsp\estsp07-training.txt
Test series file: C:\workspace\Ejemplos\Tools\xftsp\estsp07-training.txt
NRVE file: C:\workspace\Ejemplos\Tools\xftsp\nrve_10 10
Selection file: C:\workspace\Ejemplos\Tools\xftsp\selection_10 10 10
-> Step/horizon 1
Selected 3 variables: 1-3-8
Training pattern file (after selection): C:\workspace\Ejemplos\Tools\xftsp\xftsp-step-1\estsp07-training.txt-3ilo-1step---1-3-8
Test pattern file (after selection): C:\workspace\Ejemplos\Tools\xftsp\xftsp-step-1\estsp07-test.txt-3ilo-1step---1-3-8
* Performing identification (with 3 inputs) using Wang & Mendel (Active rule extraction)
Identification finished, identified 6 rules.
* Performing optimization (with 3 inputs and 6 rules) using RProp
Optimization finished
Trn MSE: 1,4906565335E-03, Tst MSE: 1,6805603718E-03 | Threshold: 1,26220182E-03 (1.15 * 1,0975668E-03)
* Performing identification (with 3 inputs) using Wang & Mendel (Active rule extraction)
Identification finished, identified 15 rules.
* Performing optimization (with 3 inputs and 15 rules) using RProp
Optimization finished
Trn MSE: 1,2759638533E-03, Tst MSE: 1,5397470334E-03 | Threshold: 1,26220182E-03 (1.15 * 1,0975668E-03)
* Performing identification (with 3 inputs) using Wang & Mendel (Active rule extraction)
Identification finished, identified 20 rules.
* Performing optimization (with 3 inputs and 20 rules) using RProp
Optimization finished
Trn MSE: 1,2574012085E-03, Tst MSE: 1,5753594329E-03 | Threshold: 1,26220182E-03 (1.15 * 1,0975668E-03)

* Results:
MF & rules & Trn. MSE & Test MSE & Trn. MxAE & Test MxAE
2 & 6 & 1,4906565335E-03 & 1,6805603718E-03 & 1,459269682E-01 & 1,5739203918E-01
3 & 15 & 1,2759638533E-03 & 1,5397470334E-03 & 1,1709453877E-01 & 1,3439983748E-01
4 & 20 & 1,2574012085E-03 & 1,5753594329E-03 & 1,2528942456E-01 & 1,4363158343E-01

Prediction: 25.098186830201954
-----
-> Step/horizon 2
```

La ejecución de *xftsp* genera asimismo una serie de directorios denominados 'xftsp-step-\*' que contienen los modelos (y ficheros auxiliares) correspondientes a cada horizonte de predicción. También se generan en estos directorios, así como en el directorio principal, otros ficheros con información sobre los sistemas generados.

### Algoritmos de identificación

En general, los algoritmos de identificación soportados por la herramienta [xfdm](#) pueden ser utilizados por *xftsp*. Algunos ejemplos son:

```
xftsp_id_algorithm(WangMendel)
xftsp_id_algorithm(ICFA, 0, 20, 2.0, 0.01, 1)
xftsp_id_algorithm(CMeans, 0, 10, 2.0, 0.01, 0 )
xftsp_id_algorithm(HardCMeans, 0, 10, 2.0, 0.01, 0 )
xftsp_id_algorithm(GustafsonKessel, 0, 10, 2.0, 0.01, 0 )
xftsp_id_algorithm(GathGeva, 0, 10, 2.0, 0.01, 0 )
xftsp_id_algorithm(IncClustering, 2, 0.1)
```

## Opciones de optimización

Conseguir una configuración adecuada de un algoritmo de optimización puede ser una tarea lenta y tediosa. A continuación se muestran algunas configuraciones que suelen funcionar bien:

```
xftsp_opt_algorithm(Scaled_conjugate_gradient)
xftsp_opt_algorithm(Rprop, 0.1, 1.5, 0.5)
xftsp_opt_algorithm(Marquardt, 0.1, 10.0, 0.2)
xftsp_opt_algorithm(Quickprop, 0.25, 1.25)
xftsp_opt_algorithm(Backprop_with_momentum, 1.2, 0.2)
xftsp_opt_algorithm(Simulated_Annealing, 500, 0.5, 100)
xftsp_opt_algorithm(Blind_search, 5.0)
xftsp_opt_algorithm(Powell, 0.5, 100)
xftsp_opt_algorithm(Simplex, 0.1, 1.5, 0.5)
```

## Ejemplo

En el directorio de ejemplos de la distribución de *Xfuzzy* se puede encontrar el fichero de configuración y los ficheros de datos necesarios para analizar una serie temporal que contiene 875 muestras semanales de temperaturas correspondientes al fenómeno “El Niño”, un patrón climático que consiste en la oscilación de los parámetros meteorológicos del Pacífico ecuatorial cada cierto número de años. Los datos se han dividido en dos subconjuntos: uno de 475 muestras, utilizado como fichero de entrenamiento, y otro con las 400 muestras restantes, empleado como fichero de test. Se ha considerado un tamaño máximo de regresor de 10 y un horizonte de predicción de 50, es decir, los 10 últimos valores conocidos serán utilizados para predecir los 50 valores siguientes.

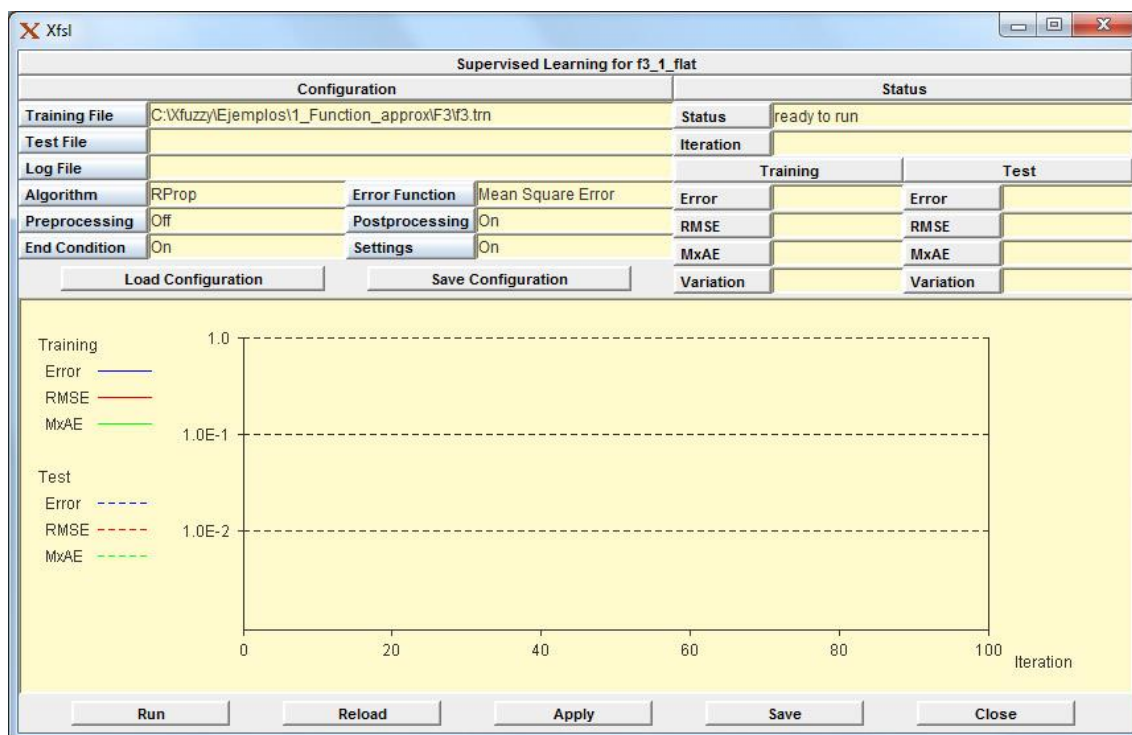
```
xftsp_series_name(estsp07)
xftsp_training_file("estsp07-training.txt")
xftsp_test_file("estsp07-test.txt")
xftsp_opt_algorithm(Rprop, 0.1, 1.5, 0.5)
xftsp_selection("selection_7")
xftsp_nrve("nrve_7")
xftsp_option(tolerance,0)
xftsp_option(max_exploration,15)
xftsp_option(generate_optimization_logs)
xftsp_option(keep_pattern_files)
```

Para realizar el estudio, lanzar la herramienta desde *Xfuzzy* cargando el fichero de configuración suministrado o ejecutar el comando:

```
$ xftsp estsp07_xftsp.cfg
```

## Herramienta de aprendizaje supervisado – Xfsl

*xfsl* es una herramienta que permite al usuario aplicar algoritmos de aprendizaje supervisado para ajustar sistemas difusos desarrollados en el flujo de diseño de *Xfuzzy 3*<sup>2</sup>. La herramienta puede ser ejecutada en modo gráfico o en modo de comando. El modo gráfico se emplea cuando se ejecuta la herramienta desde la ventana principal del entorno (usando la opción "Supervised learning" del menú *Tuning*). El modo de comando se utiliza cuando se ejecuta la herramienta desde la línea de comandos con la expresión "*xfsl file.xfl file.cfg*", donde el primer fichero contiene la definición del sistema en formato XFL3 y el segundo la configuración del proceso de aprendizaje (ver sección [Fichero de configuración](#)).



La figura anterior ilustra la ventana principal de *xfsl*. Esta ventana está dividida en cuatro partes. La parte superior izquierda corresponde a la zona utilizada para configurar el proceso de aprendizaje. El estado del proceso de aprendizaje se muestra en la parte superior derecha. La zona central muestra de forma gráfica la evolución del aprendizaje. La parte inferior de la ventana contiene varios botones de control para iniciar o parar el proceso, salvar los resultados y salir de la aplicación.

Para configurar el proceso de aprendizaje, el primer paso es seleccionar un fichero de entrenamiento que contenga los datos de entrada/salida correspondientes al comportamiento deseado. También puede seleccionarse un fichero de test cuyos datos se emplean para comprobar la capacidad de generalización del aprendizaje. El formato de ambos ficheros de patrones consiste en una serie de valores numéricos que son asignados a las variables de entrada y salida en el mismo orden que aparecen en la definición del módulo *system* de la descripción XFL3. Un ejemplo de fichero de patrones para un sistema difuso con dos entradas y una salida se muestra a continuación:

<sup>2</sup> F. J. Moreno-Velo, I. Baturone, A. Barriga, S. Sánchez-Solano  
*Automatic Tuning of Complex Fuzzy Systems with Xfuzzy*  
 Fuzzy Sets and Systems 2007  
 DOI: [10.1016/j.fss.2007.03.006](https://doi.org/10.1016/j.fss.2007.03.006)

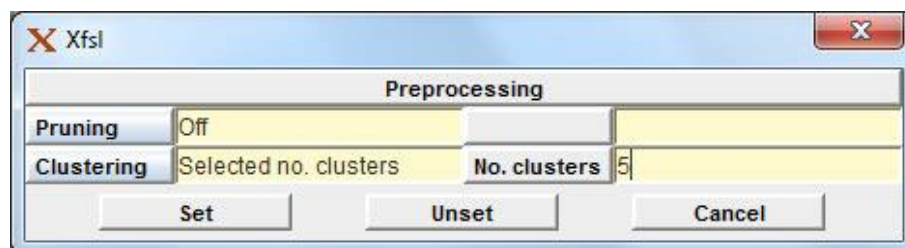
```
0.00 0.00 0.5
0.00 0.05 0.622459
0.00 0.10 0.731059
...
```

La selección de un fichero de log permite salvar la evolución del aprendizaje en un fichero externo. El empleo de este campo es opcional.

El siguiente paso en la configuración del proceso de ajuste es la selección del algoritmo de aprendizaje. *xfsi* permite el uso de muchos algoritmos diferentes (ver sección [algoritmos](#)). Entre los algoritmos básicos de descenso por el gradiente pueden seleccionarse: *Steepest Descent*, *Backpropagation*, *Backpropagation with Momentum*, *Adaptive Learning Rate*, *Adaptive Step Size*, *Manhattan*, *QuickProp* y *RProp*. Se incluyen asimismo los siguientes algoritmos de gradiente conjugado: *Polak-Ribiere*, *Fletcher-Reeves*, *Hestenes-Stiefel*, *One-step Secant* y *Scaled Conjugate Gradient*. Los algoritmos de segundo orden disponibles son: *Broyden-Fletcher-Goldfarb-Shanno*, *Davidon-Fletcher-Powell*, *Gauss-Newton* y *Mardquardt-Levenberg*. Con respecto a los algoritmos sin derivadas, pueden aplicarse: *Downhill Simplex* y *Powell's method*. Por último los algoritmos estadísticos incluidos son: *Blind Search* y *Simulated Annealing* (con esquemas de enfriamiento lineal, exponencial, clásico, rápido y adaptativo).

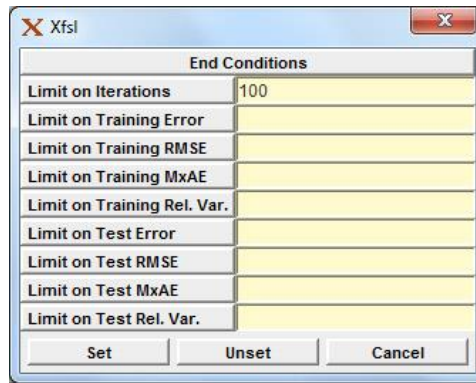
Una vez que el algoritmo ha sido seleccionado, debe elegirse una función de error. La herramienta ofrece varias funciones de error que pueden usarse para calcular la desviación entre el comportamiento actual y el deseado. (ver sección [función de error](#)). Por defecto se utiliza el error cuadrático medio (*Mean Square Error*).

*xfsi* dispone de dos algoritmos de procesado para simplificar el sistema difuso diseñado. El primero de ellos "poda" las reglas y elimina las funciones de pertenencia que no alcanzan un grado de activación o de pertenencia significativo. Existen tres variantes del algoritmo: podar todas las reglas que nunca se activan por encima de un determinado umbral, podar las N peores reglas y podar todas las reglas excepto las N mejores. El segundo algoritmo busca asociaciones o *clusters* de las funciones de pertenencia de las variables de salida. El número de clusters puede ser fijado de antemano o calculado automáticamente. Estos dos algoritmos de procesado pueden ser aplicados al sistema antes del proceso de ajuste (opción de preprocesado) o después de él (opción de postprocesado).

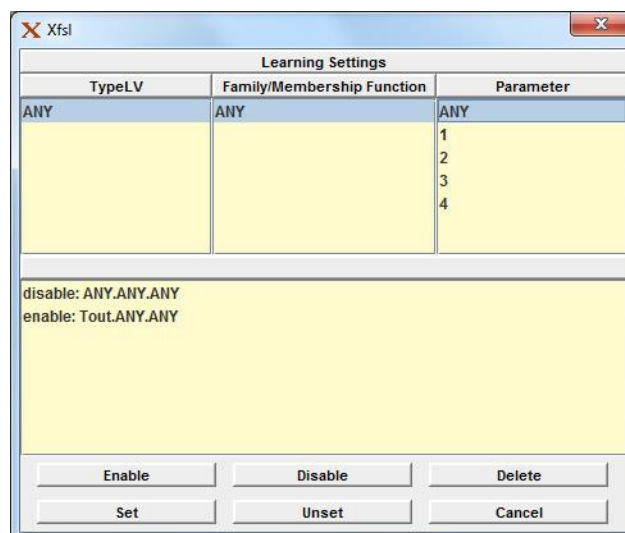


Para terminar el proceso de aprendizaje es necesario especificar una condición de fin. Dicha condición consiste en un límite impuesto sobre el número de iteraciones, el error máximo obtenido o la desviación máxima absoluta o relativa (considerando tanto los errores asociados a los datos de entrenamiento como a los de test).





La herramienta permite que el usuario seleccione los parámetros del sistema que deben ser ajustados. La siguiente ventana se utiliza para habilitar o deshabilitar el ajuste de parámetros. Las tres listas de la parte superior se emplean para seleccionar un parámetro o un conjunto de parámetros, seleccionando el tipo de la variable, la función de pertenencia de ese tipo y el índice del parámetro de esa función de pertenencia. La lista de la parte inferior de la ventana muestra la configuración actual. Las distintas líneas de configuración son interpretadas en el orden en que aparecen en la lista. En este ejemplo todos los parámetros son inicialmente deshabilitados y posteriormente se habilitan los parámetros del tipo *Tout*, de forma que sólo serán ajustados los parámetros correspondientes a este tipo.



La configuración completa del proceso de aprendizaje puede salvarse en un fichero externo que estará disponible para usos posteriores. El formato de este fichero se describe en la sección [fichero de configuración](#).

*xfsl* puede aplicarse a cualquier sistema difuso descrito por el lenguaje XFL3, incluso a sistemas que emplean funciones particulares definidas por el usuario. Lo que debe ser considerado es que las características del sistema pueden imponer limitaciones sobre los algoritmos de aprendizaje a utilizar (por ejemplo, un sistema no derivable no puede ser ajustado mediante algoritmos basados en descenso por el gradiente).

### Algoritmos

Ya que el objetivo de los algoritmos de aprendizaje supervisado consiste en minimizar una función de error que cuantifica la desviación entre el comportamiento actual y el deseado del sistema, estos algoritmos pueden ser considerados como algoritmos de optimización de



funciones. *xfs/* incluye muchos algoritmos de aprendizaje supervisado que son brevemente descritos a continuación.

#### A) Algoritmos de descenso por el gradiente

La equivalencia entre sistemas difusos y redes neuronales motivó el empleo de las técnicas de aprendizaje usadas en redes neuronales a los sistemas de inferencia difusos. En este sentido, uno de los algoritmos más conocidos empleados en sistemas difusos es el algoritmo de *BackPropagation*, que modifica los valores de los parámetros proporcionalmente al gradiente de la función de error con objeto de alcanzar un mínimo local. Ya que la velocidad de convergencia de este algoritmo es lenta, se han propuesto varias modificaciones como usar una razón de aprendizaje diferente para cada parámetro o adaptar heurísticamente las variables de control del algoritmo. Una modificación interesante que mejora en gran medida la velocidad de convergencia consiste en tener en cuenta el valor del gradiente en dos iteraciones sucesivas, lo que proporciona información sobre la curvatura de la función de error. Los algoritmos *QuickProp* y *RProp* siguen esta idea.

*xfs/* admite *Backpropagation*, *Backpropagation with Momentum*, *Adaptive Learning Rate*, *Adaptive Step Size*, *Manhattan*, *QuickProp* y *RProp*.

#### B) Algoritmos de gradiente conjugado

Los algoritmos de descenso por el gradiente generan un cambio en los valores de los parámetros que es función del valor del gradiente en cada iteración (y posiblemente en iteraciones previas). Ya que el gradiente indica la dirección de máxima variación de la función, puede resultar conveniente generar no un único paso sino varios pasos que minimicen la función de error en esa dirección. Esta idea, que es la base del algoritmo *steepest-descent*, presenta el inconveniente de producir un avance en zig-zag, porque la optimización en una dirección puede deteriorar el resultado de optimizaciones previas. La solución consiste en avanzar por direcciones conjugadas que no interfieran entre sí. Los distintos algoritmos de gradiente conjugado reportados en la literatura difieren en la ecuación utilizada para calcular las direcciones conjugadas.

El principal inconveniente de los algoritmos de gradiente conjugado es la implementación de una búsqueda lineal en cada dirección, lo que puede resultar costoso en términos de evaluaciones de la función. La búsqueda lineal puede evitarse utilizando información de segundo orden, es decir, aproximando la derivada segunda mediante dos derivadas primeras próximas. El algoritmo *scaled conjugate gradient* está basado en esta idea.

Los siguientes algoritmos de gradiente conjugado están incluidos en *xfs/*: *Steepest Descent*, *Polak-Ribiere*, *Fletcher-Reeves*, *Hestenes-Stiefel*, *One-step Secant* y *Scaled Conjugate Gradient*.

#### C) Algoritmos de segundo orden

Un paso adicional para acelerar la convergencia de los algoritmos de aprendizaje es hacer uso de información de segundo orden de la función de error, esto es, de sus derivadas segundas o, en forma matricial, de su Hesiano. Ya que el cálculo de las derivadas segundas es complejo, una posible solución es aproximar el Hesiano mediante valores del gradiente en iteraciones sucesivas. Esta es la idea de los algoritmos *Broyden-Fletcher-Goldfarb-Shanno* y *Davidon-Fletcher-Powell*.

Un caso particular importante es aquél en que la función de error a minimizar es cuadrática porque el Hesiano puede ser aproximado usando sólo las derivadas primeras de las salidas del sistema, como hace el algoritmo *Gauss-Newton*. Ya que este algoritmo puede presentar

inestabilidad cuando la aproximación del Hessian no es definida positiva, el algoritmo *Marquardt-Levenberg* resuelve este problema introduciendo un término adaptativo.

Los algoritmos de segundo orden incluidos en la herramienta son: *Broyden-Fletcher-Goldfarb-Shanno*, *Davidon-Fletcher-Powell*, *Gauss-Newton* y *Mardquardt-Levenberg*.

#### D) Algoritmos sin derivadas

No siempre es posible calcular el gradiente de la función de error, ya que dicho gradiente puede no estar definido o su cálculo puede resultar extremadamente costoso. En estos casos pueden emplearse algoritmos de optimización que no utilicen derivadas. Un ejemplo de este tipo de algoritmos es el algoritmo *Downhill Simplex*, que considera un conjunto de evoluciones de la función para decidir el cambio en los parámetros. Otro ejemplo es el *Powell's method*, que implementa búsquedas lineales mediante un conjunto de direcciones que tienden a ser conjugadas. Los algoritmos de este tipo son mucho más lentos que los anteriores. Una solución más eficiente puede ser estimar las derivadas a partir de las secantes o emplear el signo de la derivada en lugar de su valor (como hace *RProp*), el cual puede ser estimado a partir de pequeñas perturbaciones de los parámetros.

Todos los algoritmos comentados hasta el momento no alcanzan el mínimo global sino un mínimo local de la función de error. Los algoritmos estadísticos pueden descubrir el mínimo global porque generan diferentes configuraciones del sistema que expanden el espacio de búsqueda. Un modo de ampliar el espacio explorado es generar configuraciones aleatorias y elegir las mejores. Esta es la estrategia seguida por el algoritmo *Blind Search*, cuya velocidad de convergencia es extremadamente lenta. Otra alternativa consiste en realizar pequeñas perturbaciones de los parámetros hasta encontrar una solución mejor, como hacen los algoritmos de mejora iterativa. Una opción mejor es emplear algoritmos de enfriamiento simulado (*Simulated Annealing*). Estos algoritmos están basados en la analogía entre el proceso de aprendizaje, que intenta minimizar la función de error, y la evolución de un sistema físico, que tiende a disminuir su energía cuando se decrementa la temperatura. Los algoritmos de enfriamiento simulado proporcionan buenos resultados cuando el número de parámetros a ajustar es bajo. Cuando este número es alto, la velocidad de convergencia puede ser tan lenta que puede ser preferible generar configuraciones aleatorias, aplicar algoritmos de descenso por el gradiente y seleccionar la mejor solución.

Los algoritmos sin derivadas que puede aplicar *xfsi* son: *Downhill Simplex* y *Powell's method*. Los algoritmos estadísticos incluidos en la herramienta son: *Blind Search* y *Simulated Annealing* (con esquemas de enfriamiento lineal, exponencial, clásico, rápido y adaptativo).

Cuando se optimiza un sistema derivable los algoritmos *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) y *Mardquardt-Levenberg* (ML) son los más adecuados. Una buena elección para los valores de control de BFGS puede ser (0.1,10). Para el algoritmo ML los valores de control (0.1,10,0.1) son una buena elección inicial. Si no es posible calcular las derivadas del sistema, como ocurre en los sistemas jerárquicos, la mejor elección es emplear algoritmos que permitan estimar la derivada. Los algoritmos de enfriamiento simulado sólo son recomendables cuando existan pocos parámetros a ajustar y los algoritmos de segundo orden lleven al sistema hasta un mínimo local no óptimo.

### Función de error

La función de error expresa la desviación entre el comportamiento actual del sistema difuso y el deseado, comparando los patrones de entrada/salida con los valores de salida del sistema para los correspondientes valores de entrada. *xfsi* define siete funciones de error:

*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), y *classification\_square\_error* (CSE).

Todas estas funciones están normalizadas con respecto al número de patrones, al número de variables de salida y al rango de cada variable de salida, de forma que cualquier función de error puede tomar valores entre 0 y 1. Las cuatro primeras funciones son adecuadas para sistemas con variables de salida continuas, mientras que las tres últimas son específicas para sistemas de clasificación. Las ecuaciones que describen a las primeras funciones son las siguientes:

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

La salida de un sistema difuso para clasificación es la etiqueta lingüística que tiene el mayor grado de activación. La forma habitual de expresar la desviación de estos sistemas respecto al comportamiento deseado es mediante el número de fallos de clasificación (*classification\_error*, CE). Sin embargo esta elección no resulta muy adecuada para ajustar el sistema, ya que muchas configuraciones diferentes del mismo pueden producir el mismo número de fallos. Una modificación útil es añadir un término que mida la distancia entre la etiqueta seleccionada y la esperada (*advanced\_classification\_error*, ACE). Las dos funciones de error anteriores no son derivables, por lo que no pueden ser usadas con algoritmos de aprendizaje basados en derivadas (que son los más rápidos). Una elección que evita este inconveniente es considerar el grado de activación de cada etiqueta como la salida actual del sistema y la salida deseada como 1 para la etiqueta correcta y 0 para todas las demás. En este caso la función de error puede calcularse como el error cuadrático del sistema (*classification\_square\_error*, CSE), que es una función derivable con la que sí pueden usarse los algoritmos de aprendizaje basados en derivadas.

## Fichero de configuración

La configuración de un proceso de ajuste puede ser guardada en y cargada desde un fichero externo. El contenido de este fichero está formado por las siguientes directivas:

```
xfsl_training("file_name")
xfsl_test("file_name")
xfsl_log("file_name")
xfsl_output("file_name")
xfsl_algorithm(algorithm_name,value,value,...)
xfsl_option(option_name,value,value,...)
xfsl_errorfunction(function_name,value,value,...)
xfsl_preprocessing(process_name,value,value,...)
xfsl_postprocessing(process_name,value,value,...)
xfsl_endcondition(condition_name,value,value,...)
xfsl_enable(type.mf.number)
xfsl_disable(type.mf.number)
```

Las directivas *xfsl\_training* y *xfsl\_test* seleccionan los ficheros de patrones para entrenamiento y test del sistema. El fichero de log para almacenar la evolución del aprendizaje se selecciona mediante la directiva *xfsl\_log*. La directiva *xfsl\_output* contiene el nombre del fichero XFL3 donde se salvará el sistema una vez ajustado. Por defecto el nombre de este fichero es "*xfsl\_out.xfl*".

El algoritmo de aprendizaje se define con la directiva *xfs\_l\_algorithm*. Los valores se refieren a las variables de control del algoritmo. Una vez elegido el algoritmo, la directiva *xfs\_l\_option* permite seleccionar cualquiera de sus opciones específicas.

La selección de la función de error se realiza mediante la directiva *xfs\_l\_errorfunction*. Los valores contienen los pesos de las variables de salida utilizados para ponderar las funciones de error.

Las directivas *xfs\_l\_preprocessing* y *xfs\_l\_postprocessing* especifican los procesos que deben llevarse a cabo antes y después del ajuste. Las opciones posibles son: *prune\_threshold*, *prune\_worst*, *prune\_except* y *output\_clustering*. Cuando la opción *output\_clustering* contiene un valor este valor indica el número de clusters que serán creados. En caso contrario dicho número es calculado de forma automática.

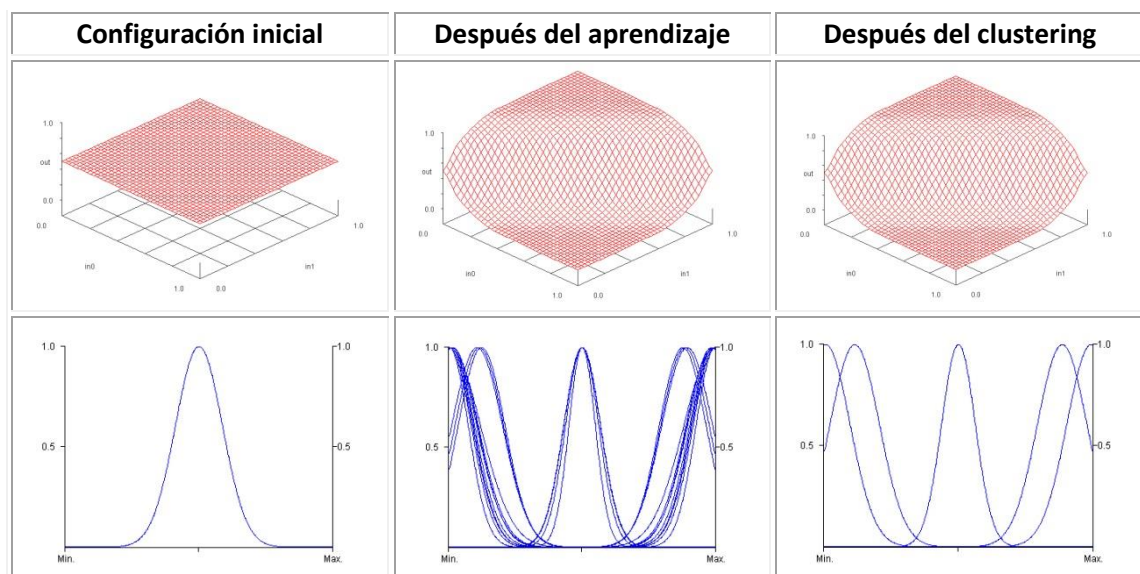
La condición de fin, seleccionada mediante *xfs\_l\_endcondition*, puede ser una de las siguientes: *epoch*, *training\_error*, *training\_RMSE*, *training\_MXAE*, *training\_variation*, *test\_error*, *test\_RMSE*, *test\_MXAE*, y *test\_variation*.

La selección de los parámetros a ajustar se realiza mediante las directivas *xfs\_l\_enable* y *xfs\_l\_disable*. Los campos *type*, *mf*, y *number* indican el tipo de la variable, la función de pertenencia y el índice del parámetro. Estos campos pueden contener también la expresión "ANY".

### Ejemplo

El directorio de ejemplos de la distribución de *Xfuzzy* contiene distintos ejemplos de procesos de ajuste. La configuración inicial del sistema se especifica en un fichero *XFL3*, que define un sistema difuso con dos variables de entrada y una de salida. Las funciones de pertenencia de las variables de salida son idénticas, de manera que el comportamiento entrada/salida de esta configuración inicial corresponde a una superficie plana.

La siguiente tabla muestra los resultados obtenidos en uno de los casos, en el que se utilizó un fichero de aprendizaje con patrones que describen la superficie dada por la expresión  $z=1/(1+\exp(10*(x-y)))$ , tras emplear el algoritmo de aprendizaje Marquardt-Levenberg y aplicar, como postprocesado, técnicas de clustering para reducir el número de funciones.

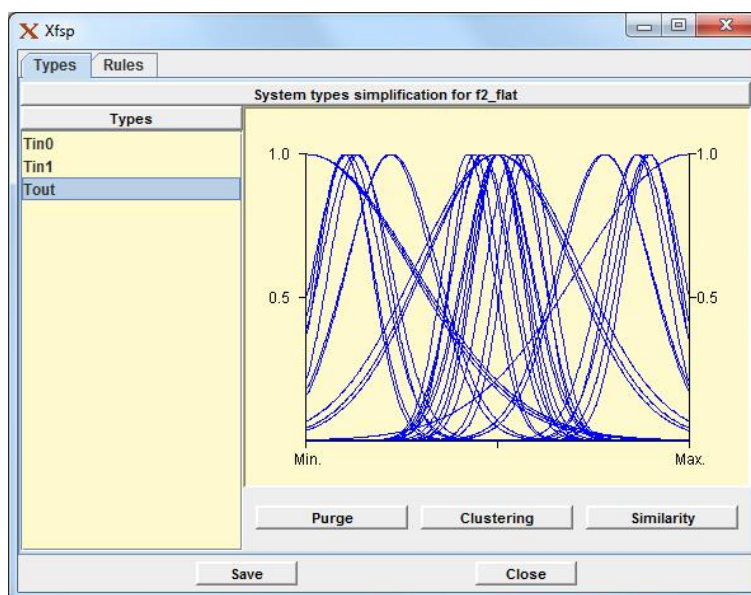


## Herramienta de simplificación - Xfsp

La herramienta *xfsp* permite aplicar algoritmos de simplificación, tanto a las funciones de pertenencia como a las bases de reglas de un sistema difuso, para obtener una descripción más sencilla o más fácilmente interpretable desde el punto de vista lingüístico<sup>3</sup>. La herramienta puede ser ejecutada mediante la opción "Simplification" del menú *Tuning* o utilizando el icono correspondiente de la ventana principal del entorno *Xfuzzy*.

### Simplificación de funciones de pertenencia

Cuando se selecciona la pestaña *Types* en la interfaz gráfica de la herramienta, las variables de entrada y salida del sistema difuso son mostradas en la parte izquierda de la ventana, mientras que las funciones de pertenencia de la variable seleccionada aparecen en la parte derecha. En esta zona se encuentran también los botones *Purge*, *Clustering* y *Similarity* que permiten aplicar los tres procesos de simplificación disponibles.



El mecanismo de purga busca y elimina las funciones de pertenencia que no son utilizadas en ninguna de las reglas. Esta circunstancia puede darse no sólo como consecuencia de procesos de simplificación previos, sino también cuando el sistema difuso ha sido definido a partir de conocimiento heurístico.

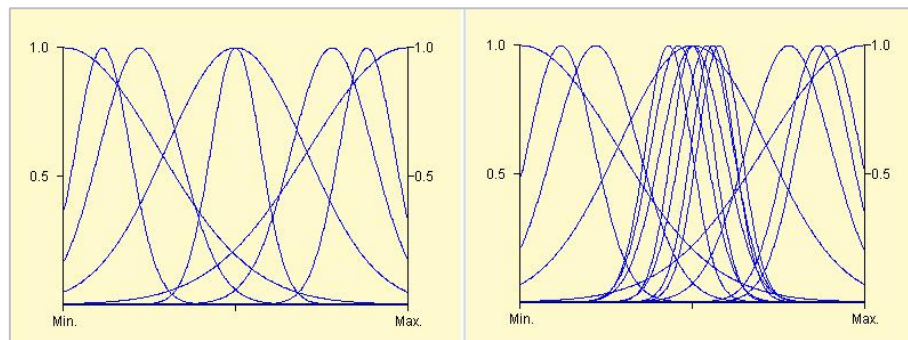
El método de agrupamiento utiliza el algoritmo Hard C-Means para buscar un número reducido de clústeres (prototipos de funciones de pertenencia) que permitan agrupar varias de las funciones originales. Los clústeres se evalúan en el espacio formado por los distintos parámetros que definen las funciones de pertenencia, siendo posible aplicar pesos a cada uno de ellos. El número final de prototipos puede ser definido por el usuario o calculado automáticamente mediante la aplicación de diferentes índices de validez: índice de separación de Dunn, índice de Davies-Bouldin e índices generalizados de Dunn.

<sup>3</sup> I. Baturone, F. J. Moreno-Velo, A. Gersnoviez  
*A CAD Approach to Simplify Fuzzy System Descriptions*  
 2006 IEEE International Conference on Fuzzy Systems  
 DOI: [10.1109/FUZZY.2006.1682033](https://doi.org/10.1109/FUZZY.2006.1682033)



La tercera técnica que contempla *xfsp* para simplificar funciones de pertenencia consiste en aplicar un proceso de fusión basado en la similitud entre las distintas funciones. Este proceso busca iterativamente el par de funciones más parecidas y las reemplaza por una función única si el grado de similitud supera un umbral definido por el usuario. El proceso finaliza cuando no se pueden fusionar más funciones.

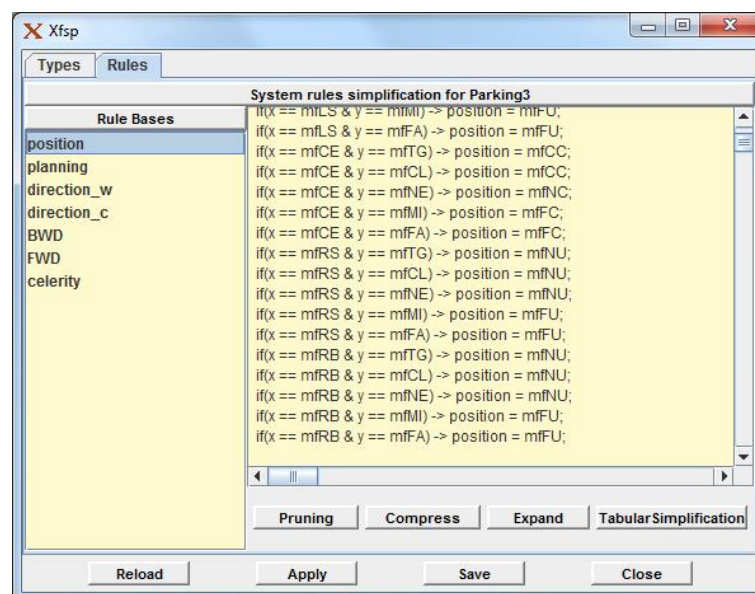
La siguiente figura muestra el resultado de aplicar diferentes procesos de simplificación de las funciones de pertenencia de la variable de salida de un sistema difuso obtenido mediante técnicas de aprendizaje supervisado.



La siguiente figura muestra el resultado de aplicar diferentes procesos de simplificación de las funciones de pertenencia de la variable de salida de un sistema difuso obtenido mediante técnicas de aprendizaje supervisado.

### Simplificación de bases de reglas

Cuando se selecciona la pestaña *Rules* en la interfaz gráfica de *xfsp*, las distintas bases de reglas que definen el comportamiento del sistema difuso son mostradas en la parte izquierda de la ventana. Al seleccionar una base de reglas, su contenido aparece en la parte derecha de la ventana, junto con los botones correspondientes a los cuatro procesos que pueden ser aplicados al conjunto de reglas: *Pruning*, *Compress*, *Expand* y *Tabular Simplification*.



El método de compresión simplemente combina todas las reglas que comparten el mismo consecuente conectando sus antecedentes mediante disyunciones (conectivo “o”). Por otra

parte, el método de expansión implementa el proceso complementario a la compresión. Ambos métodos pueden ayudar al usuario a o visualizar y comprender mejor la base de reglas, pero en realidad no realizan una simplificación efectiva. La simplificación se puede llevar a cabo realmente mediante el método de poda o el de simplificación tabular.

El proceso de poda es un método de preprocesamiento que suele aplicarse como paso previo a cualquier simplificación. Dado un conjunto de datos de entrada representativos del problema en el que se aplica el sistema de inferencia (archivo '.trn'), este proceso evalúa el grado de activación de las reglas para eliminar: (a) las n peores reglas; (b) todas las reglas excepto las n mejores reglas; o c) todas las reglas cuyo grado de activación está por debajo de un umbral. Tanto el número n como el umbral son establecidos por el usuario. La poda permite reducir el número de reglas seleccionando las más importantes en el contexto de una determinada aplicación.

El último de los mecanismos de simplificación disponibles en *xfsp* realiza una simplificación tabular de las reglas basada en una extensión del algoritmo Quine-McCluskey. Este método lleva a cabo una búsqueda lineal ordenada para encontrar todas las combinaciones de mintérminos lógicamente adyacentes de la función de n variables que se desea simplificar. Se parte de la lista de todos los mintérminos de la función para luego obtener sucesivamente listas de implicantes con (n-1), (n-2), ... variables, hasta que no sea posible formar más implicantes, obteniendo así lo que se denomina "implicantes primos de la función". El último paso consiste en seleccionar el número mínimo de implicantes primos que cubren todos los mintérminos de la función.

La siguiente figura muestra el resultado de aplicar diferentes procesos de simplificación de las bases de reglas de un sistema difuso para control de aparcamiento de un vehículo autónomo.

System rules simplification for Parking3	System rules simplification for Parking3
if(y <= mfNE & x <= mfLS) -> position = mfNU;	if(pos >= mfNC & pos <= mfCC & angle == mfRI) -> planning = mfFO;
if(y <= mfNE & x >= mfRS) -> position = mfNU;	if(pos == mfNU & angle >= mfLE & angle <= mfRI) -> planning = mfFO;
if(y >= mfMI & x >= mfRS) -> position = mfFU;	if(pos >= mfNC & pos <= mfCC & angle == mfLE) -> planning = mfFO;
if(y >= mfMI & x <= mfLS) -> position = mfFU;	if(pos >= mfNC & pos <= mfCC & angle == mfRB) -> planning = mfBA;
if(y <= mfCL & x == mfCE) -> position = mfCC;	if(angle == mfCE & pos == mfNC) -> planning = mfBA;
if(y == mfNE & x == mfCE) -> position = mfNC;	if(angle == mfCE & pos == mfCC) -> planning = mfBA;
if(y >= mfMI & x == mfCE) -> position = mfFC;	if(angle == mfCE & pos == mfFC) -> planning = mfBA;
	if(pos >= mfNC & pos <= mfCC & angle == mfLB) -> planning = mfBA;
	if(pos >= mfFC & pos <= mfFU & angle >= mfLB & angle <= mfLE) -> planning = mfPB;
	if(pos >= mfFC & pos <= mfFU & angle >= mfRI & angle <= mfRB) -> planning = mfPB;
	if(pos == mfFU & angle >= mfLB & angle <= mfRB) -> planning = mfPB;

## Etapa de síntesis

La etapa de síntesis es el último paso en el flujo de diseño de un sistema. Su objetivo es generar una implementación del sistema que pueda ser usada externamente. Existen dos tipos diferentes de implementaciones finales para sistemas difusos: implementaciones software e implementaciones hardware. La síntesis software genera la representación del sistema en un lenguaje de programación de alto nivel. La síntesis hardware genera un circuito microelectrónico que implementa el proceso de inferencia descrito por el sistema difuso.

Las implementaciones software resultan útiles cuando no existen fuertes restricciones sobre la velocidad de inferencia, el tamaño del sistema o el consumo de potencia. Este tipo de implementación puede ser generada a partir de cualquier sistema difuso desarrollado en *Xfuzzy*. Por otra parte, las implementaciones hardware son más adecuadas cuando se requiere alta velocidad o bajo consumo de área y potencia, pero, para que esta solución sea eficiente, es necesario imponer ciertas restricciones sobre el sistema difuso, de forma que la síntesis hardware no es tan genérica como la alternativa software.

*Xfuzzy 3* proporciona al usuario tres herramientas para síntesis software: [xfc](#), que genera una descripción del sistema en ANSI-C; [xfcpp](#), para generar una descripción C++; y [xfj](#), que describe el sistema difuso mediante una clase Java. En cuanto a las facilidades de síntesis hardware, el entorno incluye la herramienta [xfvhdl](#) que genera una descripción VHDL sintetizable basada en una arquitectura específica para sistemas difusos, y la herramienta [xfsg](#) que genera un modelo Simulink que puede ser implementado sobre FPGAs utilizando las herramientas de desarrollo de DSP de Xilinx (SysGen).

### Herramienta de generación de código ANSI-C – Xfc

La herramienta *xfc* genera una representación del sistema difuso en código ANSI-C. La herramienta puede ser ejecutada desde la línea de comandos, con la expresión "*xfc file.xfl [output\_dir]*", o desde el menú *Synthesis* de la ventana principal del entorno. Ya que la representación ANSI-C no necesita ninguna información adicional, esta herramienta no implementa una interfaz gráfica de usuario específica; sólo aparecerá una ventana que permite seleccionar el directorio en el que se almacenarán los ficheros generados.

A partir de la especificación de un sistema difuso en formato XFL3, *systemname.xfl*, la herramienta genera dos ficheros: *systemname.h*, que contiene la definición de las estructuras de datos; y *systemname.c*, que contiene las funciones C que implementan el sistema de inferencia difuso.

Para un sistema difuso con variables de entrada globales *i0*, *i1*, ..., y variables de salida globales *o0*, *o1*, ..., la función de inferencia incluida en el fichero *systemname.c* es:

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

La función que realiza la inferencia puede ser utilizada en proyectos C externos incluyendo en ellos el fichero de cabecera (*systemname.h*).



## Herramienta de generación de código C++ - Xfcpp

La herramienta *xfcpp* genera una representación C++ del sistema difuso. Puede ser ejecutada desde la línea de comandos, con la expresión "*xfcpp file.xfl [output\_dir]*", o a través del menú *Synthesis* de la ventana principal de Xfuzzy. Esta herramienta tampoco tiene una interfaz gráfica de usuario específica porque la generación de la representación C++ no necesita información adicional; sólo aparecerá una ventana para seleccionar el directorio de salida.

A partir de la especificación de un sistema difuso en formato XFL3, *systemname.xfl*, la herramienta genera cuatro ficheros: *xfuzzy.hpp*, *xfuzzy.cpp*, *systemname.hpp* y *systemname.cpp*. Los ficheros *xfuzzy.hpp* y *xfuzzy.cpp* contienen la descripción de las clases C++ que son comunes a todos los sistemas difusos. Los ficheros *systemname.hpp* y *systemname.cpp* contienen la descripción de las clases específicas del sistema *systemname.xfl*. Los ficheros con extensión '.hpp' son ficheros de cabecera que definen las estructuras de las clases, mientras que los ficheros con extensión '.cpp' contienen el cuerpo de las funciones de cada clase. Todos los ficheros son generados en el directorio *output\_dir*, indicado al ejecutar la herramienta (por defecto, el mismo donde reside el fichero *systemname.xfl*).

El código C++ generado por *xfcpp* implementa un motor de inferencia difuso que puede ser utilizado con valores crisp y con valores difusos. Un valor difuso se encapsula en un objeto de clase *MembershipFunction*.

```
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() {}
};
```

La clase que define el sistema difuso es una extensión de la clase abstracta *FuzzyInferenceEngine*. Esta clase, definida en *xfuzzy.hpp*, contiene cuatro métodos que implementan el proceso de inferencia difuso.

```
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;
};
```

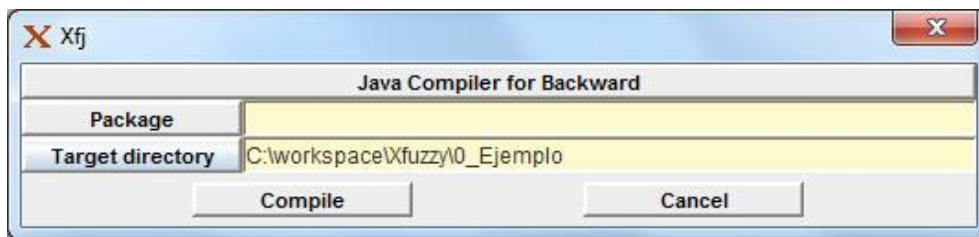
El fichero *systemname.cpp* contiene la descripción de la clase *systemname*, que implementa el proceso de inferencia difuso del sistema. Además de describir los cuatro métodos de la clase *FuzzyInferenceEngine*, la clase del sistema contiene un método, llamado *inference*, que implementa el proceso de inferencia con variables en lugar de con arrays de variables. La función de inferencia para un sistema difuso con variables de entrada globales *i0*, *i1*, ..., y variables de salida globales *o0*, *o1*, ..., es:

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



## Herramienta de generación de código Java – Xfj

La herramienta *xfj* genera una representación Java del sistema difuso. La herramienta puede ser ejecutada desde la línea de comandos, con la expresión *xfj [-p package] file.xfl [output dir]*, o desde el menú *Synthesis* de la ventana principal del entorno. Cuando se invoca desde la línea de comandos no aparece interfaz gráfica. En este caso los ficheros con código Java se generan en el directorio de salida especificado al ejecutar la herramienta (o en el directorio que contiene al fichero del sistema, si no se indica otra cosa) y se añade una instrucción *package* en las clases Java cuando se usa la opción *-p*. Cuando *xfj* es invocada desde la ventana principal de *Xfuzzy* el nombre del *package* y el directorio de destino pueden ser elegidos a través de la interfaz gráfica de la herramienta.



A partir de la especificación de un sistema difuso en formato *XFL3*, *systemname.xfl*, la herramienta genera cuatro ficheros: *FuzzyInferenceEngine.java*, *MembershipFunction.java*, *FuzzySingleton.java* y *systemname.java*. Los tres primeros ficheros corresponden a las descripciones de dos interfaces y una clase que son comunes a todos los sistemas de inferencia difusos. El último fichero contiene la descripción específica del sistema difuso.

El fichero *FuzzyInferenceEngine.java* describe una interfaz Java que define un sistema de inferencia difuso general. Esta interfaz define cuatro métodos para implementar el proceso de inferencia con valores crisp y difusos.

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

El fichero *MembershipFunction.java* contiene la descripción de una interfaz usada para describir un número difuso. Contiene sólo un método, llamado *compute*, que calcula el grado de pertenencia para cada valor del universo de discurso del número difuso.

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

La clase *FuzzySingleton* implementa la interfaz *MembershipFunction*, que representa un valor crisp como un número difuso.

```
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); }
}
```

Finalmente, el fichero *systemname.java* contiene la clase que describe el sistema difuso. Esta clase es una implementación de la interfaz *FuzzyInferenceEngine*. Por tanto, los métodos públicos que implementan la inferencia son los de la interfaz (*crispInference* y *fuzzyInference*).

## Herramienta de síntesis software – Xfsw

*xfsw* proporciona un comando unificado para las herramientas de generación de código C, C++ y Java. Sólo puede utilizarse desde la línea de comandos usando el siguiente formato:

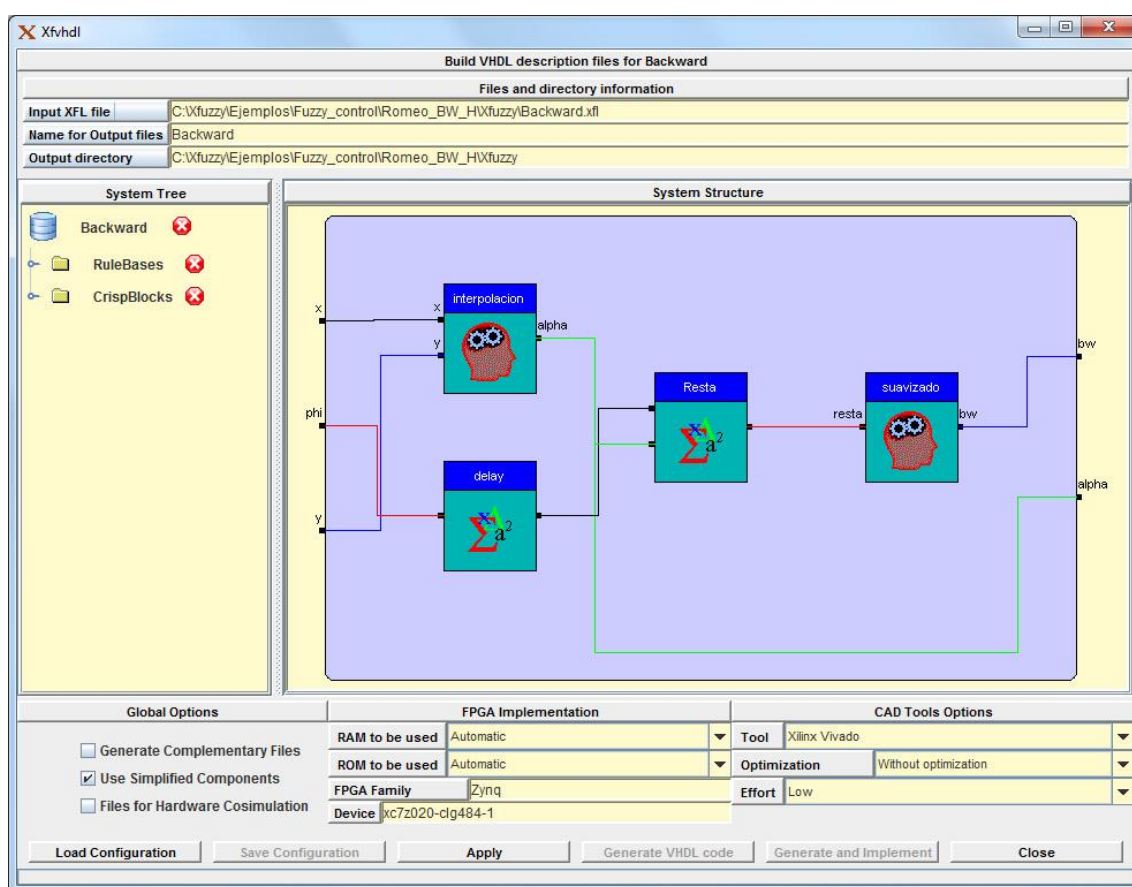
```
xfsw (-ansic|-c++|-java [-p package_name]) file.xfl [output_dir]
```

Los parámetros son equivalentes a los utilizados por cada herramienta de forma individual. El directorio en el que se generan los ficheros es el indicado en el parámetro *output\_dir* o, en su defecto, la ruta en la que se encuentra el fichero *.xfl*.

## Herramienta de generación de código VHDL – Xfvdhl

La herramienta *xfvdhl* utiliza el lenguaje de descripción de hardware de alto nivel VHDL para facilitar la implementación hardware, mediante FPGAs o ASICs, de los sistemas de inferencia descritos en el entorno *Xfuzzy*<sup>4</sup>. Una característica importante de esta herramienta es que permite la síntesis directa de sistemas difusos complejos, formados por la combinación de distintos módulos de inferencia y bloques crisp. Sin embargo, no todas las especificaciones XFL3 son susceptibles de ser implementadas en hardware mediante *xfvdhl*. En concreto, los sistemas difusos que pueden ser implementados por esta herramienta deben usar funciones de pertenencia con grado de solapamiento máximo 2 y emplear métodos de defuzzificación simplificados.

La interfaz gráfica de *xfvdhl* puede ser ejecutada desde la ventana principal del entorno, utilizando la opción "To VHDL" del menú *Synthesis*, o desde la línea de comandos, mediante la expresión "*xfvdhl -g file.xml [file.xml]*".



La ventana principal de *xfvdhl* está dividida en cuatro partes. La zona superior recoge información sobre los ficheros y directorios involucrados en el diseño. El campo *Input XFL file* contiene la ruta absoluta del fichero de especificación XFL3 de entrada seleccionado al lanzar la herramienta. Este campo es sólo informativo, es decir, no es modificable por el usuario. El campo *Name for output files* permite configurar el prefijo de los ficheros de salida de *xfvdhl*.

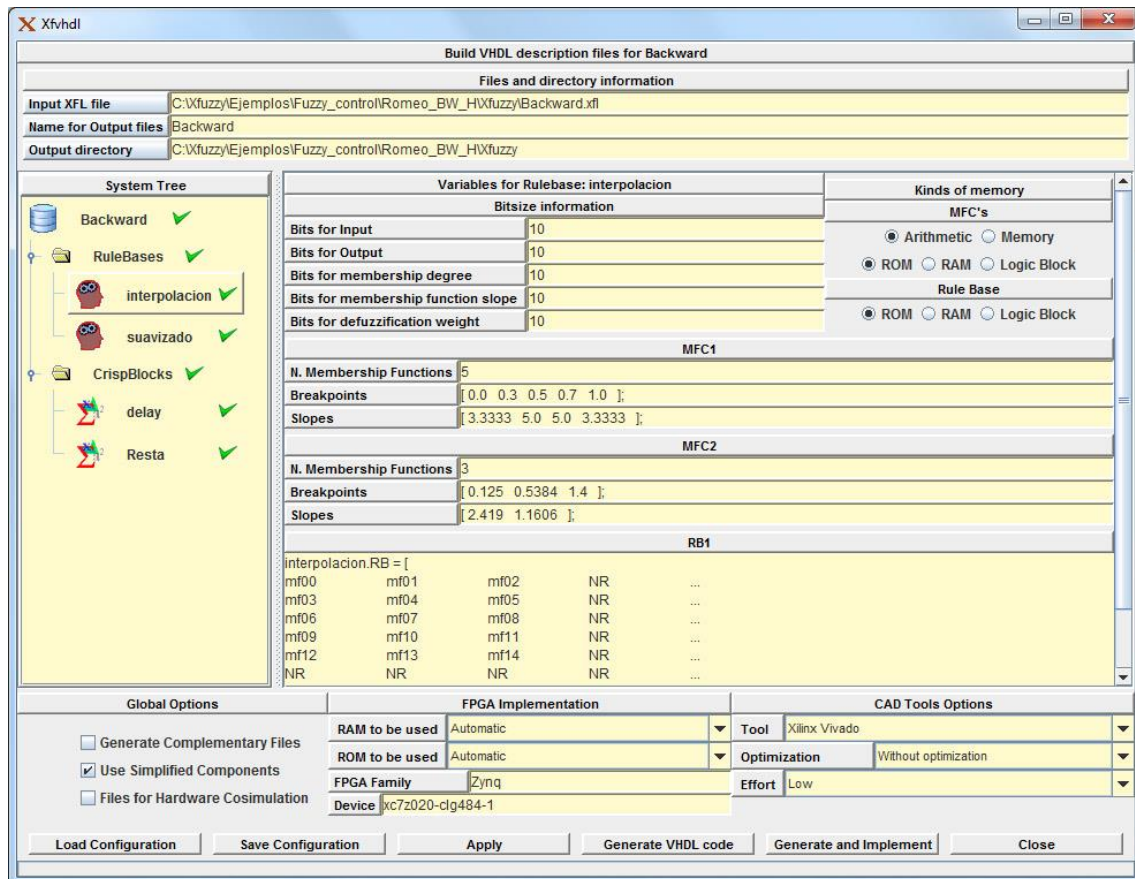
<sup>4</sup> M. Brox, S. Sánchez-Solano, E. del Toro, P. Brox, F. J. Moreno-Velo  
*CAD Tools for Hardware Implementation of Embedded Fuzzy Systems on FPGAs*  
 IEEE Transactions on Industrial Informatics 2012  
 DOI: [10.1109/TII.2012.2228871](https://doi.org/10.1109/TII.2012.2228871)

Por defecto aparece el nombre del sistema difuso de entrada, aunque puede ser modificado por el usuario. Finalmente, el campo *Output directory* indica la ruta absoluta del directorio donde se ubicarán los ficheros de salida que genera la herramienta. Su valor por defecto es el directorio que contiene la especificación del sistema.

La zona inferior de la ventana contiene tres secciones que permiten definir diferentes opciones de síntesis e implementación. En la sección *Global Options* el usuario puede indicar que se generen ficheros complementarios marcando la opción *Generate Complementary Files*. También puede indicar que se usen componentes simplificados a través de la opción *Use Simplified Components*. En el caso de que se elija esta opción, en la descripción VHDL se incluirá la versión simplificada (sin bloque de división) para los defuzzificadores Fuzzy Mean y Takagi-Sugeno, siempre que la especificación del sistema lo permita (sistemas con funciones de pertenencia normalizadas que empleen el operador producto como conectivo de antecedentes; la herramienta obviará el uso de componentes simplificados en los casos en que no se verifiquen estas condiciones aunque la opción aparezca seleccionada). Por último, cuando se marca la opción *Files for Hardware Cosimulation*, la herramienta genera descripciones VHDL de salida adaptadas para ser incorporadas en modelos Simulink mediante el empleo de "Black Boxes". La sección *FPGA Implementation* recoge información relativa a opciones de implementación para FPGAs. Entre ellas se encuentra el tipo de memoria RAM y ROM a usar (inicialmente aparece la opción *Automatic* en ambas, aunque en un menú desplegable también se pueden seleccionar las opciones *None*, *Block* o *Distributed*), así como la familia de FPGAs y el dispositivo sobre el que se quiere implementar el sistema de inferencia (por defecto aparece Zynq xc7z020-clg484-1). Finalmente, la sección *CAD Tool Options* hace referencia a una serie de opciones relacionadas con las herramientas de CAD. Entre ellas se encuentran: la herramienta de síntesis a utilizar (la opción por defecto es *Xilinx Vivado*, aunque también se puede seleccionar *Xilinx XST*); el tipo de optimización (la opción preseleccionada es *Without optimization*, pero en el menú se pueden marcar las opciones *Area optimization*, *Speed optimization* y *Area and Speed optimizations*); y el esfuerzo con el que se lleva a cabo la síntesis (a priori está seleccionada la opción *Low*, aunque también puede marcarse la opción *High* en el menú desplegable).

La zona central de la ventana está a su vez dividida en dos partes. Inicialmente, en la derecha aparece la representación gráfica de la especificación XFL3, mientras que en la izquierda se muestran los distintos componentes de la base de conocimiento estructurados en forma de árbol y agrupados bajo las categorías *RuleBases* y *CrispBlocks*. Cuando se selecciona una base de reglas concreta, el contenido de la zona central derecha es sustituido por una nueva interfaz que permite al usuario definir los parámetros relacionados con la dimensión del sistema. Concretamente se puede introducir el número de bits con el que se codifican las entradas, la salida, los grados de pertenencia de los antecedentes, las pendientes de las funciones de pertenencia y el parámetro de peso del método de defuzzificación (en los casos en que este exista). También en esta zona el usuario puede seleccionar la estrategia de implementación de los antecedentes (en memoria o mediante cálculo aritmético) y el tipo de memoria utilizada (ROM, RAM o bloque lógico). La herramienta permite la generación de funciones de pertenencia normalizadas de tipo *triangular*, *sh\_triangular*, y *trapezoid* mediante técnicas aritméticas. En el caso de que las funciones de pertenencia de las entradas no estén normalizadas, se inhabilita la opción de cálculo aritmético en los antecedentes. Para la memoria de reglas también puede elegirse si se quiere implementar con memoria ROM, RAM o mediante un bloque lógico. En la parte inferior de esta zona se muestra información extraída de la especificación XFL3 relativa a las funciones de pertenencia y las bases de reglas. Concretamente esta zona incluye los valores del número de funciones de pertenencia, puntos de corte y pendientes por cada entrada, así como la representación matricial de la base de

reglas correspondiente. Los valores mostrados sólo tienen carácter informativo, por lo que no pueden ser modificados.



Al seleccionar un bloque crisp dentro de la estructura de árbol, en la zona central derecha aparece un solo campo relativo al número de bits con el que se codifica la salida del bloque.

Cuando todas las opciones arquitecturales y los parámetros relacionados con el tamaño de los buses correspondientes a una base de reglas han sido definidos, debe asignarse esta configuración mediante el botón *Apply* (localizado en la parte inferior de la ventana). Tras ello, el icono rojo que aparecía junto a la base de conocimiento en la primera figura se sustituye por el icono verde que se observa en la segunda. Una vez que la información correspondiente a todas las bases de reglas y bloques crisp del sistema ha sido definida, el componente asociado al sistema difuso también es identificado con una marca verde y se activan los botones *Save Configuration*, *Generate VHDL code* y *Generate and Implement*.

El botón *Save Configuration* permite guardar la configuración del sistema mediante un fichero XML que almacena información relativa a las opciones de implementación de los distintos componentes (ver sección [Fichero de configuración](#)). Las configuraciones guardadas mediante esta opción pueden ser cargadas posteriormente utilizando el botón *Load Configuration*. o utilizadas para ejecutar la herramienta en modo no interactivo (ver sección [Ejecución en modo de comando](#)).

## Ficheros de salida

El botón *Generate VHDL code* genera la descripción VHDL del sistema difuso junto con un fichero de *testbench*, descrito también en VHDL, que permite verificar su funcionalidad. La descripción VHDL del sistema se genera en un único fichero compuesto por la interconexión de



bloques de la librería de celdas *XfuzzyLib*. La cabecera de este fichero incluye también un paquete de constantes calculadas de forma automática a partir de la información extraída de la base de conocimiento del sistema de inferencia y de los parámetros y opciones de diseño introducidos por el diseñador. Si el sistema es jerárquico se genera una descripción VHDL por cada base de reglas, así como un *testbench* que permite obtener la superficie de control correspondiente a cada una de ellas. En este caso se generan también el fichero VHDL correspondiente al nivel superior de la jerarquía (*top-level*), que describe la interconexión de las distintas bases de reglas y bloques críps que forman el sistema, y un *testbench* que permite simular el sistema completo.

Además de los ficheros anteriores, si la herramienta de síntesis seleccionada es *Xilinx Vivado*, se generan dos ficheros de comandos con extensión “.tcl”. El fichero “<spec>.tcl” facilita la creación de un proyecto Vivado para llevar a cabo las tareas de verificación e implementación del sistema. “<spec>Script.tcl” permite automatizar los procesos de síntesis e implementación del sistema difuso utilizando las herramientas de Xilinx en modo no-proyecto.

Cuando la herramienta seleccionada es *Xilinx XST*, se generan dos ficheros adicionales con extensiones “.prj” y “.xst”. El fichero “<spec>.prj” contiene la relación de los módulos del sistema. “<spec>Script.xst” contiene comandos que dirigen el proceso de síntesis con la herramienta XST, muchos de los cuales son independientes de las opciones elegidas, y otros dependen de las mismas (concretamente, los comandos *rom\_extract* y *ram\_extract* dependen de las opciones elegidas en el tipo de ROM y RAM a usar en el campo *FPGA implementation*).

Finalmente, si se ha seleccionado la opción de generar ficheros complementarios, se obtiene una serie de ficheros con extensiones “.dat”, “.dat.bin” y “.plt”. Estos ficheros contienen información relacionada con el contenido de las memorias de antecedentes y las bases de reglas del sistema para su posterior estudio. Se generan un fichero “.dat” y otro “.dat.bin” por cada variable de entrada, que contienen los datos de las memorias de antecedentes (combinaciones de valores de etiqueta-grado-grado) en decimal y en binario, respectivamente. El fichero “.plt” es un fichero de comandos de *Gnuplot* que permite representar gráficamente las funciones de pertenencia. Por último, se obtiene un fichero con extensión “.dat” que incluye el contenido de la memoria de reglas.

Durante la creación de los ficheros se puede producir algún error o warning que se comunicará al usuario en el área de mensajes de *Xfuzzy*. El listado de los errores, junto a la descripción de las causas que los motivan, se ilustra en la sección [mensajes de error](#).

El botón *Generate and Implement* genera los mismos ficheros que el botón *Generate VHDL code*, pero además sintetiza el código VHDL y lo implementa sobre la FPGA de Xilinx especificada en *FPGA implementation*, con las opciones de implementación especificadas en *Cad Tools Options*, haciendo uso de las herramientas de síntesis e implementación de Xilinx. En esta fase puede aparecer el mensaje “*There are errors, so can't execute any synthesis tool*” o “*There are errors, so can't execute any implementation tool*” si se ha producido previamente algún error en la etapa de creación de ficheros.

## Ejecución en modo de comando

La herramienta *xfvhdl* también se puede ejecutar desde un terminal utilizando los siguientes comandos:

- *xfvhdl -g <XFL3> [<XML>]*: Permite abrir la interfaz gráfica de *xfvhdl* cargando la especificación XFL3 de un sistema difuso. En caso de que se especifique un archivo de configuración XML también se carga el fichero de configuración indicado.

- *xfvdl* <XFL3> [<XML>] [options]: Genera el código VHDL para la especificación XFL3 con el fichero de configuración XML. El campo [options] admite los siguientes modificadores:
  - S: (genera código VHDL y sintetiza)
  - I: (genera código VHDL, sintetiza e implementa)
  - L <library>: (usa la librería VHDL indicada, en lugar de usar la definida por defecto).

## Fichero de configuración

La configuración del proceso de síntesis con *xfvhd* puede guardarse en un fichero XML. La raíz del fichero de configuración es la etiqueta denominada *system*, que tiene tres atributos: *name*, *rulebases* y *crisps*. El primero indica el nombre del sistema, mientras que los otros dos indican, respectivamente, el número de bases de reglas y de bloques crisp.

```
<?xml version="1.0" encoding="UTF-8" ?>
<system name="Backward" rulebases="2" crisps="1">
  <rulebases>
    <rulebase name="interpolacion" inputs="2" outputs="1">
      <bits_input>8</bits_input>
      <bits_output>8</bits_output>
      <bits_membership_degree>8</bits_membership_degree>
      <bits_MF_slopes>8</bits_MF_slopes>
      <bits_def_weight>8</bits_def_weight>
      <MFC_arithmetic>true</MFC_arithmetic>
      <MFC_memory>ROM</MFC_memory>
      <RB_memory>ROM</RB_memory>
    </rulebase>
    <rulebase name="suavizado" inputs="1" outputs="1">
      <bits_input>9</bits_input>
      <bits_output>9</bits_output>
      <bits_membership_degree>9</bits_membership_degree>
      <bits_MF_slopes>2</bits_MF_slopes>
      <bits_def_weight>9</bits_def_weight>
      <MFC_arithmetic>true</MFC_arithmetic>
      <MFC_memory>ROM</MFC_memory>
      <RB_memory>ROM</RB_memory>
    </rulebase>
  </rulebases>
  <crisps>
    <crisp name="Resta" inputs="2" outputs="1">
      <bitsize_output>9</bitsize_output>
    </crisp>
  </crisps>
  <options>
    <complementary_files>>false</complementary_files>
    <use_simp_components>true</use_simp_components>
    <hardware_cosimulation>>false</hardware_cosimulation>
    <FPGA_RAM>0</FPGA_RAM>
    <FPGA_ROM>0</FPGA_ROM>
    <FPGA_family>Zynq</FPGA_family>
    <FPGA_device>xc7z020-clg484-1</FPGA_device>
    <CAD_tool>0</CAD_tool>
    <CAD_optimization>0</CAD_optimization>
    <CAD_effort>0</CAD_effort>
    <outputFile>Backward</outputFile>
    <outputDirectory>C:\Xfuzzy\Ejemplo\OUT</outputDirectory>
  </options>
</system>
```



El fichero incluye tres elementos principales: *rulebases*, *crisps* y *options*. La etiqueta *rulebases* contiene información sobre las bases de reglas, cada una de ellas identificada con la etiqueta *rulebase*. Este elemento tiene como atributos: *name*, que indica el nombre de la base de reglas; *inputs*, que indica el número de entradas; y *outputs*, que indica el número de salidas. Los elementos hijos de esta etiqueta definen cada uno de los parámetros de la base de reglas: *bits\_input* (número de bits para las entradas), *bits\_output* (número de bits para las salidas), *bits\_membership\_degree* (número de bits para el grado de pertenencia), *bits\_MF\_slopes* (número de bits para las pendientes), *bits\_def\_weight* (número de bits para el peso de los defuzzificadores que utilizan este parámetro), *MFC\_arithmetic* (booleano que indica si se ha escogido implementar los MFCs mediante circuitos aritméticos (*true*) o mediante una memoria (*false*)), *MFC\_memory* (indica el tipo de memoria escogida para la memoria de antecedentes) y *RB\_memory* (indica el tipo de memoria escogida para la memoria de reglas).

El elemento *crisps* aparece vacío cuando el sistema no incluye ningún bloque de este tipo. En caso contrario, cada bloque es definido mediante una etiqueta *crisp* que incluye los atributos: *name*, que indica el nombre del bloque; *inputs*, que indica el número de entradas; y *outputs*, que indica el número de salidas. El único parámetro que puede definirse para este tipo de elementos es el número de bits con el que se codifica la salida (*bitsize\_output*).

Por último, la etiqueta *options* se utiliza para identificar las distintas opciones que aparecen en la parte inferior de la interfaz gráfica de *xfvhdl*. Los elementos hijos de dicha etiqueta son: *complementary\_files* (booleano que indica si el usuario selecciona la opción de generación de ficheros complementarios), *use\_simp\_components* (booleano que muestra si el usuario selecciona la opción de usar métodos de defuzzificación simplificados), *FPGA\_RAM* (número del 0 al 3 que indica el tipo de memoria RAM utilizada; 0=*Automatic*, 1=*None*, 2=*Block*, 3=*distributed*), *FPGA\_ROM* (número del 0 al 3 que indica la opción elegida para la memoria ROM utilizada; 0=*Automatic*, 1=*None*, 2=*Block*, 3=*distributed*), *FPGA\_family* (texto que indica la familia de FPGAs escogida por el usuario), *CAD\_tool* (número 0 o 1 que indica la herramienta de síntesis escogida 0=*Xilinx Vivado*, 1=*Xilinx XST*), *CAD\_optimization* (número del 0 al 3 que indica la optimización que se quiere usar; 0=*Without optimization*, 1=*Area optimization*, 2=*Speed optimization*, 3=*Area and Speed optimization*), y *CAD\_effort* (número 0 o 1 que indica el esfuerzo que se quiere usar; 0=*Low*, 1=*High*).

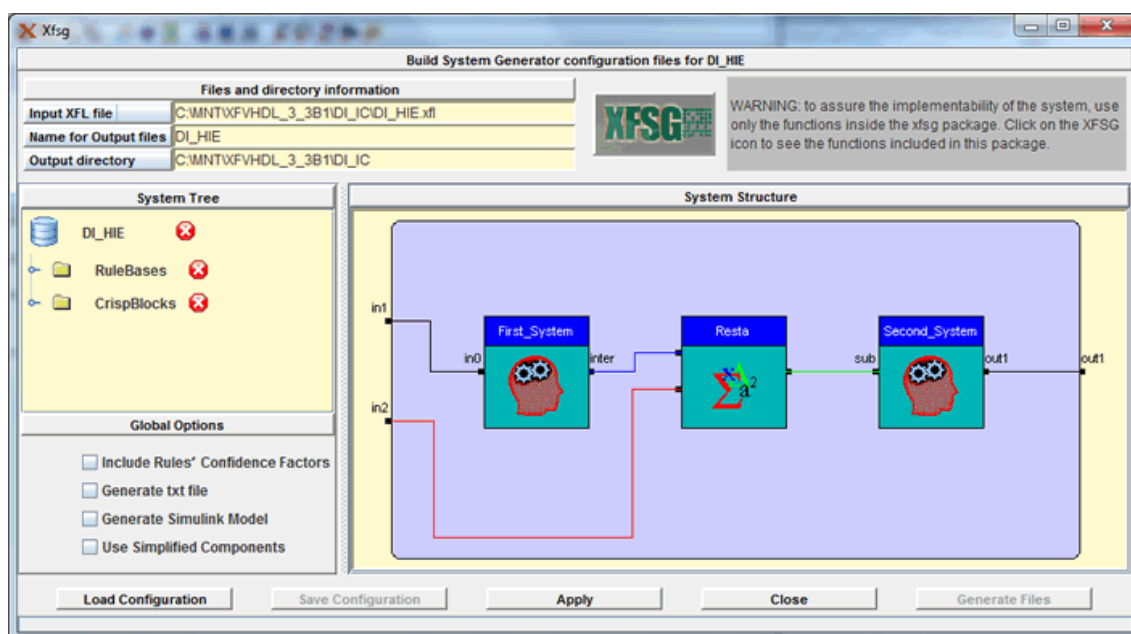
## Mensajes de error

Error	Descripción
<i>Can't create output directory</i>	Aparece cuando hay un fallo al crear alguno de los ficheros de salida
<i>The maximum overlapping degree must be two in variable &lt;i&gt;</i>	Se produce cuando en cierta variable hay un solapamiento distinto de 2 que no está permitido en la arquitectura en la que se basa la herramienta
<i>There isn't any membership function in variable &lt;i&gt;</i>	Indica que no se han definido funciones de pertenencia para la variable <i>
<i>It is not allowed rulebases with more tan two inputs and Takagi-Sugeno as defuzzification method: &lt;rulebase-name&gt;</i>	Aparece cuando se ha utilizado un sistema con más de dos entradas y Takagi-Sugeno como defuzzificador
<i>Error in rule: &lt;FLC-name&gt;</i>	Se produce cuando hay un fallo en una regla de un módulo de inferencia
<i>It is not allowed rulebases with more tan one output: &lt;rulebase-name&gt;</i>	Se produce cuando la base de reglas tiene más de una salida
<i>No prefix file valid. By default &lt;OUTPUT_FILE_DEFAULT&gt;</i>	Indica que se utiliza el prefijo por defecto en la generación de los ficheros de salida debido a que el introducido no es válido
<i>AND operation not valid. Will be used Minimum by default</i>	Indica que va a ser utilizado el conectivo Mínimo porque el operador AND que se ha utilizado no está soportado
<i>Families of Membership Functions not allowed</i>	Se produce cuando se usan funciones de pertenencia o familias de funciones de pertenencia que no están soportadas
<i>The xml file is not correctly defined</i>	Aparece cuando se utiliza un archivo XML erróneo
<i>Exception in defuzzification method: &lt;FLC-name&gt;</i>	Aparece cuando se produce alguna incompatibilidad entre la herramienta y el defuzzificador usado en el módulo de inferencia
<i>The bitsize for membership function slope is too short, you must resize it or choose memory for the MFCs in &lt;FLC-name&gt;</i>	Se produce cuando no se han asignado suficientes bits para codificar las pendientes de las funciones de pertenencia

## Herramienta de generación de modelo SysGen – Xfsg

La herramienta de síntesis hardware *xfsg* (*Xfuzzy to System Generator*) permite trasladar de manera automática la especificación XFL3 de un sistema difuso jerárquico, compuesto por la combinación de distintos módulos de inferencia y bloques crisp, en un modelo Simulink que puede ser simulado en el entorno MATLAB e implementado en FPGAs de Xilinx<sup>5</sup>. Sin embargo, no todas las especificaciones XFL3 son susceptibles de ser implementadas mediante *xfsg*. En concreto, los sistemas difusos que pueden ser implementados por esta herramienta deben emplear funciones o familias de funciones de pertenencia triangulares con grado de solapamiento 2 y usar métodos de defuzzificación simplificados.

La interfaz gráfica de *xfsg* puede ser ejecutada desde la ventana principal del entorno *Xfuzzy*, utilizando la opción "To Sysgen" del menú *Synthesis*, o a través del icono correspondiente de la barra de botones. La ventana principal de *xfsg* está dividida en cinco partes: una zona con información sobre la localización y el nombre de los ficheros utilizados, la estructura en árbol que desglosa las bases de reglas y bloques crisp que componen el sistema, un área en la que inicialmente se muestra la interconexión de los distintos componentes del sistema, una zona de opciones globales, y una serie de botones situados en la parte inferior de la ventana.



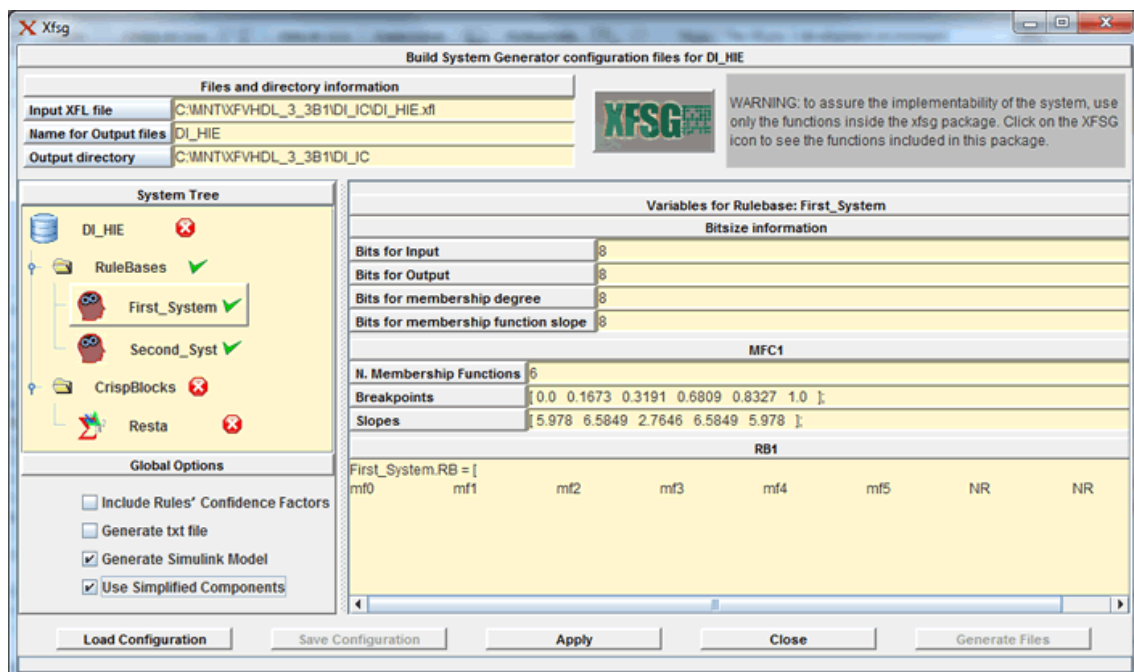
La zona de información sobre ficheros y directorios está dividida en tres campos. El campo *Input XFL file* contiene la ruta absoluta del fichero de especificación XFL3 seleccionado al lanzar la herramienta. Este campo es informativo, no puede ser modificado por el usuario. El campo *Name for Output files* permite configurar el prefijo de los ficheros de salida de *xfsg*. Por defecto aparece el nombre del sistema difuso de entrada. Finalmente, el campo *Output directory* indica la ruta absoluta del directorio donde se ubicarán los ficheros de salida que genera la herramienta. En este caso aparece por defecto el directorio que contiene la especificación del sistema.

<sup>5</sup> S. Sánchez-Solano, E. del Toro, M. Brox, P. Brox, I. Baturone  
*Model-Based Design Methodology for Rapid Development of Fuzzy Controllers on FPGAs*  
 IEEE Transactions on Industrial Informatics 2012  
 DOI: [10.1109/TII.2012.2211608](https://doi.org/10.1109/TII.2012.2211608)

La zona superior de la ventana también incluye un botón (identificado por el texto XFSG) que, al ser pulsado, muestra un cuadro de diálogo donde se enumeran los distintos operadores, métodos de defuzzificación, tipos de funciones de pertenencia y bloques crisp que pueden aparecer en los sistemas difusos sintetizados por la herramienta. Estas funciones se encuentran definidas en lo que en la terminología de *Xfuzzy* se denomina el “paquete *xfsg*”. A la derecha del botón aparece un texto que aconseja al usuario usar solo las funciones incluidas en dicho paquete para asegurar que no se produzca ningún problema al implementar el sistema.

En la zona central izquierda de la ventana se muestra la estructura del sistema difuso en forma de árbol, con los elementos que lo componen agrupados bajo las categorías *RuleBases* y *CrispBlocks*. Inicialmente, o siempre que se seleccione el nivel superior de la especificación del sistema, en la zona de la derecha aparece una ventana con los componentes que forman el sistema y su interconexión. Cuando se selecciona una base de reglas concreta dentro de la categoría *RuleBases*, aparece en esta zona la interfaz que se muestra en la siguiente figura. En ella el usuario puede definir los distintos parámetros que dimensionan el módulo de inferencia. En concreto se puede introducir el número de bits con el que se codifican las entradas, la salida, el grado de pertenencia de los antecedentes y las pendientes de las funciones de pertenencia. También en esta zona se pueden visualizar ciertos valores calculados a partir de la especificación del sistema. Concretamente, el número de funciones de pertenencia y los valores de los puntos de corte y las pendientes por cada entrada, así como la representación matricial de la base de reglas correspondiente.

Cuando se selecciona un bloque crisp en la estructura de árbol, la parte central derecha de la interfaz muestra un solo campo a rellenar relativo al número de bits definidos para la salida del bloque.



Cuando todos los parámetros relativos a la base de reglas o al bloque crisp han sido configurados es necesario pulsar el botón *Apply* para guardar los cambios realizados (en caso contrario se perderá la información introducida en el formulario). Tras ello, el icono rojo que aparecía inicialmente junto a la base de conocimiento se sustituye por el icono verde que se observa en la figura. Cuando los parámetros de todas las bases de reglas y bloques crisp que

componen el sistema han sido definidos, aparece también un icono verde junto al nivel superior de la especificación del sistema y se habilitan los botones *Save Configuration* y *Generate Files* de la zona inferior de la interfaz.

El botón *Save Configuration* permite guardar la configuración del sistema mediante un fichero XML que almacena información relativa a las opciones de implementación de los distintos componentes del sistema (ver sección [Fichero de configuración](#)). Las configuraciones guardadas mediante esta opción pueden ser cargadas en un momento posterior utilizando el botón *Load Configuration*.

Antes de pulsar el botón *Generate Files* el usuario puede configurar las opciones que aparecen en la zona *Global Options* de la interfaz gráfica. La funcionalidad de cada una de las opciones es la siguiente:

- *Include Rule's Confidence Factors*: Al activar esta opción se incluirá en el fichero de salida “.m”, para cada una de las bases de reglas del sistema, una matriz con los grados de certeza de las reglas. Esta opción está contemplada en el lenguaje de especificación XFL3 aunque de momento no se utiliza para implementaciones hardware de sistemas de inferencia.
- *Generate txt file*: Al activarla se creará un fichero “.txt” que contiene información textual sobre la estructura del sistema.
- *Generate Simulink model*: Si esta opción está activada se creará el fichero “.mdl” correspondiente al modelo Simulink del sistema difuso.
- *Use Simplified Components*: Si se activa esta opción se usarán componentes simplificados siempre que sea posible, es decir, cuando el método de defuzzificación sea *Fuzzy Mean* o *Takagi-Sugeno*, el conectivo de antecedentes sea el operador producto y la base de reglas esté completamente especificada.

## Ficheros de salida

Una vez definidos los parámetros de los distintos componentes del sistema y las opciones globales, se puede pulsar el botón *Generate Files* para generar los siguientes ficheros en el directorio de salida indicado:

- *<FLC>.m* es un fichero “.m” de MATLAB que contiene la inicialización de las variables de cada uno de los bloques de la librería *XfuzzyLib* que se usan para implementar el sistema difuso. Este fichero siempre se genera con independencia de las opciones escogidas en la zona *Global Options*.
- *<FLC\_aux>.mdl* contiene el modelo Simulink del sistema difuso utilizando los módulos incluidos en la librería *XfuzzyLib*.
- *<FLC>.txt* contiene una descripción en formato texto de las entradas y salidas de cada base de reglas y bloque crisp. También incluye el componente que se utiliza de la librería *XfuzzyLib*. Si no existe dicho componente, se especifica con *null*.

## Fichero de configuración

La configuración del proceso de síntesis con *xfsg* puede guardarse en un fichero XML para ser recuperada en un momento posterior. Debe tenerse en cuenta que la sintaxis del fichero de configuración puede cambiar en las sucesivas versiones de *Xfuzzy* y que sólo se pueden cargar

ficheros de configuración generados con la versión actual, por lo que los ficheros XML antiguos se deben adaptar al formato correcto agregando las nuevas etiquetas.

La apariencia del fichero de configuración refleja la estructura en forma de árbol que representa al sistema. La raíz de dicho fichero es la etiqueta denominada *system*, que tiene tres atributos: *name*, *rulebases* y *crisps*. El primero indica el nombre del sistema, mientras que los otros dos indican el número de bases de reglas y de bloques crisps, respectivamente. (Si el sistema no contiene ningún bloque crisp no aparece este atributo).

El fichero incluye tres elementos principales *rulebases*, *crisps* y *options*. La etiqueta *rulebases* contiene información sobre las bases de reglas, cada una de ellas identificada con la etiqueta *rulebase*. Este elemento tiene como atributos: *name*, que indica el nombre de la base de reglas; *inputs*, que indica el número de entradas; y *outputs*, que indica el número de salidas. Los elementos hijos de esta etiqueta definen cada uno de los parámetros de la base de reglas: *bits\_input* (número de bits para las entradas), *bits\_output* (número de bits para las salidas), *bits\_membership\_degree* (número de bits para el grado de pertenencia) y *bits\_MF\_slopes* (número de bits para las pendientes).

El elemento *crisps* aparece vacío cuando el sistema no incluye ningún bloque de este tipo. En caso contrario, cada bloque es definido mediante una etiqueta *crisp* que incluye los atributos *name*, que indica el nombre del bloque, *inputs*, que indica el número de entradas, y *outputs*, que indica el número de salidas. El único parámetro que puede definirse para este tipo de elementos es el número de bits con el que se codifica la salida (*bitsize\_output*).

```
<?xml version="1.0" encoding="UTF-8"?>
<system name="Backward" rulebases="2" crisps="2">
  <rulebases>
    <rulebase name="interpolacion" inputs="2" outputs="1">
      <bits_input>10</bits_input>
      <bits_output>10</bits_output>
      <bits_membership_degree>10</bits_membership_degree>
      <bits_MF_slopes>10</bits_MF_slopes>
    </rulebase>
    <rulebase name="suavizado" inputs="1" outputs="1">
      <bits_input>10</bits_input>
      <bits_output>10</bits_output>
      <bits_membership_degree>10</bits_membership_degree>
      <bits_MF_slopes>10</bits_MF_slopes>
    </rulebase>
  </rulebases>
  <crisps>
    <crisp name="delay" inputs="1" outputs="1">
      <bitsize_output>10</bitsize_output>
    </crisp>
    <crisp name="Resta" inputs="2" outputs="1">
      <bitsize_output>10</bitsize_output>
    </crisp>
  </crisps>
  <options>
    <include_rule_confidence_factor_mfile>false</include_rule_confidence_factor_mfile>
    <gen_txtfile>false</gen_txtfile>
    <gen_simmodel>true</gen_simmodel>
    <use_simp_components>true</use_simp_components>
    <outputFile>Backward</outputFile>
    <outputDirectory>C:\Xfuzzy\examples\Tools\xfsg\OUT</outputDirectory>
  </options>
</system>
```

Por último, la etiqueta *options* se utiliza para identificar las distintas opciones que aparecen en las secciones *Global Options* y *Files and directory information* de la interfaz gráfica de *xfvhdI*. Los elementos hijos de dicha etiqueta son: *include\_rule\_confidence\_factor\_mfile*, *gen\_txtfile*, *gen\_simmodel*, *use\_simp\_components*, *outputFile* y *outputDirectory*. Los cuatro primeros admiten un valor booleano (*true* o *false*) que indica la activación o no de la correspondiente opción.

Las configuraciones guardadas pueden ser posteriormente cargadas utilizando el botón *Load Configuration*, sin necesidad de introducir nuevamente todos los valores.

### Mensajes de error

Si se produce algún error o warning durante la generación de los ficheros de salida de *xfsg*, se comunicará al usuario en el área de mensajes de Xfuzzy. El listado de los posibles errores junto a la descripción de las causas que los motivan se ilustra en la siguiente tabla.

Error	Descripción
<i>Can't create output directory</i>	Aparece cuando la herramienta no puede crear el directorio indicado como salida
<i>There isn't a Simulink component to this rulebase. You must creat it !!!</i>	Se produce cuando no existe una arquitectura tipo dentro de <i>XfuzzyLib</i> para implementar una de las bases de reglas del sistema
<i>You can't use a simplified component</i>	Se produce cuando se ha seleccionado la opción <i>Use Simplified Components</i> , pero una base de reglas no puede utilizar el componente simplificado
<i>Invalid membership function to calculate the weight of the rules</i>	Aparece cuando se usa el defuzzificador <i>Weighted Fuzzy Mean</i> y en la definición de las funciones de pertenencia de salida falta el segundo parámetro característico de estos métodos
<i>Membership functions incorrect for inputs</i>	Aparece cuando se usa un tipo de función de pertenencia no permitida. La herramienta admite triángulos libres normalizados y familias de triángulos, donde el primer y/o el último elemento pueden ser trapecios
<i>The rulebase is not complete</i>	Se produce cuando no está definido el consecuente para todas las posibles combinaciones de las etiquetas de las entradas
<i>Invalid name system, Invalid name rulebase, Invalid name crisp</i>	Se produce cuando se carga un fichero de configuración y los nombres de las bases de reglas, bloques crisp o del sistema no se corresponden con los que aparecen en la especificación en <i>Xfuzzy</i>
<i>Invalid rule</i>	Indica que una regla incluye algún operador que no se ha tenido en cuenta dentro de la herramienta

## Historial de revisiones

<b>Fecha</b>	<b>Versión</b>	<b>Descripción</b>
11/03/2018	3.5_00	Documentación Xfuzzy_3.5