Facultad de Informática

Universidad Complutense de Madrid

Proyecto de Sistemas Informáticos
Curso 2006-2007

FuSiE (Fuzzy Systems Editor)

Realizado por: Adoración GONZALEZ TOBOSO
Reyes Lucio PERUCHA SÁNCHEZ
María Luisa URBINA GARCÍA

Dirigido por: Prof. D. Francisco Javier CRESPO YÁÑEZ

Madrid, Septiembre de 2007

DEDICATORIA

A nuestras familias, sin las que nada de esto habría sido posible.

AGRADECIMIENTOS

Hemos de manifestar nuestro agradecimiento a algunas de las personas que nos han ayudado a llevar adelante este proyecto:

A Francisco Javier Crespo Yánez, por la atención y dedicación con que ha dirigido el proyecto.

A Pablo Gervás Gómez-Navarro, por colaborar en la realización de este trabajo.

Y, en general, a todo aquél que nos ha prestado su apoyo durante la realización del proyecto.

A todos, gracias.

AUTORIZACIÓN

Autorizamos a la Universidad Complutense a difundir este documento, así como el código que hemos elaborado, con fines meramente académicos, y nunca comerciales, y haciendo mención en todos los casos de sus autores.

Adoración GONZALEZ TOBOSO
Reyes Lucio PERUCHA SÁNCHEZ
María Luisa URBINA GARCÍA

RESUMEN

En este trabajo se realiza el diseño de una herramienta para modelizar sistemas basados en lógica borrosa. Dicho diseño presenta numerosas ventajas, carentes algunas de ellas en herramientas consultadas en la literatura al efecto. Varias de esas ventajas son: reusabilidad, comprehensión de los datos, y flexibilidad frente a los cambios y mejoras que surgen durante el periodo de explotación. Además, se ha formalizado el diseño creado mediante un prototipo experimental, que contiene la funcionalidad básica del sistema diseñado.

ABSTRACT

In this work we realize the design of a tool that are able to build, use and maintain fuzzy systems. This design presents numerous advantages, lacking some of them in tools consulted in the literature to the effect. Several of these advantages are: reusability, and flexibility in case of changes and improvements that arise during the period of development. In addition, there has been formalized the design created by means of an experimental prototype, which contains the basic functionality of the designed system.

PALABRAS CLAVE

Lógica borrosa, modelado de sistemas borrosos, razonamiento monotónico, herramienta borrosa, diseño sistemas borrosos, métodos de inferencia.

KEY WORDS

Fuzzy logia, fuzzy systems modeling, monotonic reasoning, fuzzy tool, fuzzy systems design, inference method.

ÍNDICE GENERAL

1. INTRODUCCIÓN 12

1.1. PLANTEAMIENTO DEL PROBLEMA 13
1.2. OBJETIVOS DEL PROYECTO 13
1.2.1. Objetivo general 13
1.2.2. Objetivos particulares 14
1.3. APORTACIONES 14
1.4. METODOLOGÍA 15
1.5. MÉTODO DE EVALUACIÓN 17

2. ESTADO DE LA CUESTIÓN 18

2.1. INTRODUCCIÓN A LA LÓGICA BORROSA 18
2.1.1. Conjuntos borrosos 19
2.1.2. Funciones de Pertenencia 20
2.1.3. Características de un Conjunto Borroso 24
2.1.4. Operaciones Unarias sobre Conjuntos Borrosos 25
2.1.5. Relaciones entre Conjuntos Borrosos 25
2.2. OPERACIONES CON CONJUNTOS BORROSOS 26
2.2.1. Propiedades Básicas 27
2.2.2. Normas y Conormas Triangulares 28
2.2.3. Operaciones de agregación 29
2.2.4. Negaciones de conjuntos borrosos 31
2.3. VARIABLES LINGÜÍSTICAS 31
2.3.1. Modificadores lingüísticos 33
2.3.2. Cuantificadores Lingüísticos 35
2.4. LÓGICA BORROSA Y SISTEMAS BASADOS EN REGLAS 36
2.4.1. Cálculos con Lógica Borrosa 37
2.4.2. Razonamiento o Inferencia 39
2.4.3. Sistemas Basados en Reglas 39
2.4.4. Sintaxis de las Reglas Borrosas 41
2.4.5. Cálculos de las reglas difusas 43
2.5. MODELOS BORROSOS 46
2.5.1. Fases del Modelado de Sistemas 46
2.5.2. Topología del modelado borroso 47
2.6. CONTROL BORROSO 48
2.6.1. Características de los Controladores Difusos [Sur, Omron, 1997] 50

3. ESTUDIO Y RESOLUCIÓN DEL PROBLEMA 52

3.1. MODELO PROPUESTO 52
3.2. FASES DEL PROYECTO 54
3.2.1. Fase I : Estudio de las Estructuras de la Aplicación 55
3.2.2. Fase II : Estudio de las Funciones del Código de la Aplicación 60
3.2.3. Fase III : Estudio del Código de la Aplicación 64
3.2.4. Diseño obtenido de las fases anteriores 68
3.2.5. Fase IV: Diseño de las clases e interfaces 69

CONCLUSIONES 100

LÍNEAS FUTURAS 103

BIBLIOGRAFÍA 104

1. INTRODUCCIÓN

La lógica borrosa nace de la teoría de los conjuntos borrosos del profesor Zadeh. El objetivo del profesor Zadeh era expresar en lenguaje matemático el modo de razonamiento aproximado, característico de los humanos. Contrariamente a la lógica tradicional en la que solo son posibles dos valores, en la lógica borrosa todo es cuestión de grado. El grado es el concepto fundamental de la lógica borrosa. Las cosas ya no son blancas o negras, ahora los tonos grises están permitidos [APR05].

Aunque la intención original del profesor Zadeh al introducir la Teoría del razonamiento aproximado, era crear un formalismo para manipular la imprecisión y vaguedad del razonamiento humano expresado lingüísticamente, causó sorpresa que el éxito de la lógica borrosa llegase al campo del control automático de procesos [ROB07].

Hasta ahora, han sido un gran número de productos los lanzados al mercado usando lógica borrosa. Dicha lógica ha sido probada para ser particularmente útil en el control de sistemas: control de tráfico, control de vehículos (helicópteros...), control de compuertas en plantas hidroeléctricas, centrales térmicas, control en máquinas lavadoras, control de metros (mejora de su conducción, precisión en las paradas y ahorro de energía), ascensores...; en la predicción y optimización: predicción de terremotos, optimizar horarios...; en el reconocimiento de patrones y visión por ordenador: seguimiento de objetos con cámara, reconocimiento de escritura manuscrita, reconocimiento de objetos, compensación de vibraciones en la cámara...; y para sistemas de información o conocimiento: Bases de datos, sistemas expertos...; entre otros muchos campos.

El éxito de estas aplicaciones de control radica en la simplicidad, tanto conceptual como de desarrollo de los procesos de control borrosos [ROB07].

1.1. Planteamiento del problema

Los sistemas basados en lógica borrosa constituyen una importante alternativa a tener en cuenta en numerosas aplicaciones. En el campo del control automático de procesos es donde las aplicaciones de lógica borrosa adquieren su mayor relevancia. Es preciso el desarrollo de sistemas que permitan la implementación, el diseño, y el aprendizaje de sistemas basados en lógica borrosa orientados a aplicaciones de control [UGR06]. En los últimos años son numerosas las aplicaciones que se han desarrollado para el modelado de sistemas basados en lógica borrosa, muchas de estas aplicaciones están limitadas en cuanto a la reusabilidad del código, comprehensión de los datos, y compactación del sistema.

Se presenta el desarrollo software de una herramienta multiplataforma, que permite el modelado de sistemas basados en lógica borrosa, además de contar con las ventajas mencionadas anteriormente: reusabilidad del código, comprehensión de los datos, y compactación del sistema.

1.2. Objetivos del proyecto

1.2.1. Objetivo general

El objetivo general es desarrollar una herramienta multiplataforma para modelizar fácilmente sistemas basados en lógica borrosa, que presente características tales como reusabilidad, comprehensión de los datos y flexibilidad frente a cambios y mejoras que surgirán durante el periodo de explotación.

1.2.2. Objetivos particulares

Los objetivos particulares del trabajo son:

- Realizar un diseño para el prototipo de la herramienta, documentándolo mediante diagramas de clases que aportarán mayor claridad e información, de modo que facilitará la labor de futuros desarrollos.

- Desarrollar casos de uso, que facilitarán la comprensión del funcionamiento de la herramienta, y servirán como método de comparación con las demás herramientas de modelado de sistemas borrosos, consultadas en la literatura al efecto.

1.3. Aportaciones

Como paso previo a la realización de nuestra herramienta FuSiE (Fuzzy Systems Editor), realizaremos un proceso de reringeniería partiendo de la herramienta de Earl Cox. De este paso obtendremos un diseño de la herramienta a estudio, así como un resumen detallado de las funciones principales del sistema

Para facilitar el entendimiento del funcionamiento de la herramienta de Earl Cox al lector, aportamos el diseño, y resumen detallado de las funciones principales del sistema que obtendremos en el proceso de reingeniería.

1.4. Metodología

La metodología seguida para la realización de la herramienta será la aplicación de programación orientada a objetos, para dotar al sistema de las ventajas que ésta programación aporta. Entre estas ventajas podemos destacar, uniformidad para la aplicación y los módulos que la compongan, comprehensión, estabilidad e integridad de los datos, evitando redundancias y ambiguedades en su uso, así como flexibilidad para la implementación de nuevas mejoras, hacer la aplicación más mantenible y la posibilidad de reutilizar código y partes del sistema.

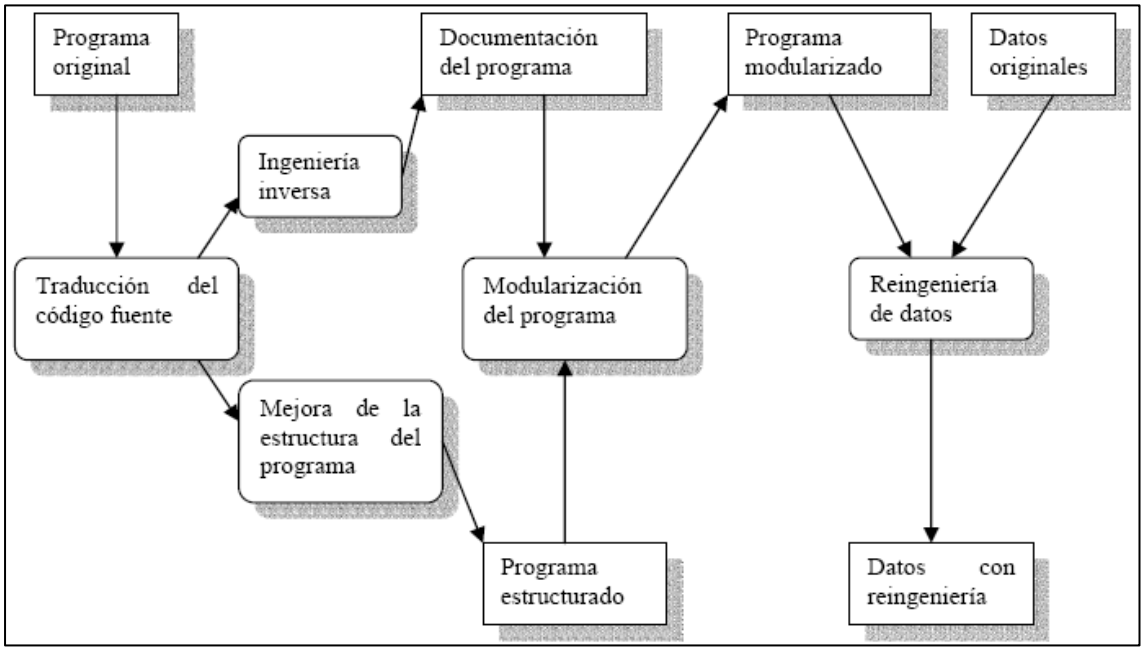
El desarrollo de esta herramienta se basa en el estudio de aplicaciones de modelización de sistemas borrosos creadas con anterioridad. Para ello usaremos un proceso de reingeniería sobre la herramienta de Cox. La elección de esta herramienta y no otras está motivada porque antes de comenzar con el proyecto habíamos estudiado documentación de Earl Cox acerca de sistemas borrosos, y nos parecía apropiado el enfoque que el autor hacía para el desarrollo de nuestra herramienta.

Hemos optado por este proceso para nuestro trabajo, teniendo en cuenta la propia definición de lo que es la reingeniería, re-estructurar o re-escribir una parte o todo, de un sistema heredado sin cambiar su funcionalidad [Gervas05].

El problema se debe a que no existe un único proceso establecido de reingeniería software, ya que las particularidades de cada proyecto pueden requerir unas tareas u otras dentro del proceso [MRS05].

Después de comparar algunos de los procesos de reingeniería existentes, optamos por aplicar a nuestro proyecto una adaptación de las tareas propuestas por Sommerville. Hemos llevado a cabo dicha elección motivados porque las tareas que propone Sommerville son las que mejor se amoldan a los objetivos establecidos en el proyecto.

Según Sommerville, la reingeniería de software comprende: la redocumentación del sistema, la reorganización y reestructura del sistema, la traducción del sistema a un lenguaje de programación más moderno, la modificación y actualización de la estructura y los valores de los datos del sistema. El proceso de reingeniería de Sommerville, más completo, sería el que se recoge en la figura [Gervas05].



El proceso que seguiremos para la consecución del trabajo consta de las siguientes fases:

1. Fase I: Estudio de las Estructura de la Aplicación.
2. Fase II: Estudio de las Funciones del Código de Aplicación.
3. Fase III: Estudio del Código de Aplicación.
4. Fase IV: Diseño de clases e interfaces.
5. Fase V: Modularización del programa.

Las tres primeras fases, se identificarán con la fase definida por Sommerville como Fase de Ingeniería Inversa. Durante cada una de estas tres fases, analizaremos el programa creado por Earl Cox y extraeremos información, la cual nos ayuda a documentar la organización y funcionalidad. Es el proceso de analizar el software con el objetivo de recuperar su diseño y especificación.

La cuarta fase la identificaremos con la Fase de Mejora de la Estructura del Programa, de Sommerville. En ella realizaremos el diseño de la herramienta, para ello aplicaremos el conocimiento obtenido del análisis realizado en s fases anteriores.

Y la última fase, la identificaremos con la fase que tiene el mismo nombre en el proceso propuesto por Sommerville. En ella, reorganizaremos el programa de forma que las partes relacionadas se integren de forma conjunta.

1.5. Método de Evaluación

La evaluación de la consecución de los objetivos planteados en el apartado *1.2. Objetivos del proyecto*, está orientada a la comprobación de las ventajas que el diseño de la herramienta FuSie aporta a las herramientas en las que hemos basado el estudio, en concreto de la herramienta de Earl Cox.

Las ventajas que hay que observar en FuSiE se deben a que el diseño facilita la reusabilidad del código, lleva a una aplicación mantenible, flexible frente a mejoras que puedan aparecer en la fase de explotación; a la vez que se ha logrado una compactación del sistema. Además, la herramienta FuSiE contará con las ventajas de una aplicación multiplataforma.

2. ESTADO DE LA CUESTIÓN

2.1. Introducción a la lógica borrosa

La palabra *fuzzy* viene del ingles *fuzz* (tamo, pelusa, vello) y se traduce por difuso o borroso[1]. Lotfi A. Zadeh es el padre de toda esta teoría **[Zadeh, 1965]** y en la actualidad es un campo de investigación muy importante, tanto por sus implicaciones matemáticas o teóricas como por sus aplicaciones prácticas.

Muchos conceptos que manejamos los humanos a menudo, no tienen una definición clara como por ejemplo ¿Qué es una persona alta? ¿A partir de qué edad una persona deja de ser joven?. La lógica clásica o bivaluada es demasiado restrictiva, una afirmación puede no ser ni VERDADERA (*true*) ni FALSA (*false*). “Yo leeré El Quijote”: ¿En qué medida es cierto? Depende de quien lo diga y...

Es recomendable usar la tecnología borrosa o fuzzy **[Sur, Omron, 1997]** cuando haya que introducir la experiencia de un operador “experto” que se base en conceptos imprecisos obtenidos de su experiencia. Cuando ciertas partes del sistema a controlar son desconocidas y no pueden medirse de forma fiable (con errores posibles). Cuando el ajuste de una variable puede producir el desajuste de otras. En general, cuando se quieran representar y operar con conceptos que tengan imprecisión o incertidumbre (como en las Bases de Datos Borrosas).

[1] Utilizaremos ambas palabras, difuso y borroso, como sinónimas.

2.1.1. Conjuntos borrosos

Los conjuntos borrosos surgen como una nueva forma de representar la imprecisión y la incertidumbre. Es una herramienta útil en campos de las Matemáticas, Probabilidad, Estadística, Filosofía, Psicología... En cambio los conjuntos clásicos (crisp) surgen de forma natural, por la necesidad del ser humano de clasificar objetos y conceptos.

Los conjuntos clásicos se definen mediante la función de pertenencia $A(x)$, $x \in X$ donde X es el universo del discurso y la restricción de la función es A: X→{0,1}.

– Conjunto Vacío → Vacio(x)=0, para todo $x \in X$
– Conjunto Universo → U(x)=1, para todo $x \in X$

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

Los conjuntos borrosos(fuzzy) relajan la restricción, A: X →[0,1] debido a la existencia de conceptos que no tienen límites claros como ¿La temperatura 25ºC es "alta"?, en este caso definimos, por ejemplo: Alta(30)=1, Alta(10)=0, Alta(25)=0.75...

Un conjunto borroso A se define como una Función de Pertenencia que enlaza o empareja los elementos de un dominio o Universo de discurso X con elementos del intervalo [0,1]:

$$A: X \rightarrow [0,1]$$

Cuanto más cerca esté $A(x)$ del valor 1, mayor será la pertenencia del objeto x al conjunto A. Los valores de pertenencia varían entre 0 (no pertenece en absoluto) y 1 (pertenencia total).

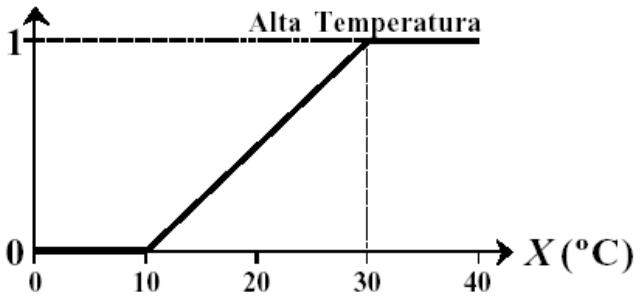
Un conjunto borroso ***A*** puede representarse como un conjunto de pares de valores: Cada elemento ***x€X*** con su grado de pertenencia a ***A***. También puede ponerse como una "suma" de pares**:**

- $A = \{ A(x)/x,\ x \in X \}$
- $A = \sum_{i=1}^{n} A(x_i)/x_i$ os pares en los que ***A*(*xi*)=0**, no se incluyen)

El contexto es fundamental en la definición de conjuntos borrosos ya que no es lo mismo el concepto "Alto" aplicado a personas que a edificios. Un conjunto borroso puede representarse también gráficamente como una función, especialmente cuando el universo de discurso *X* (o dominio subyacente) es continuo (no discreto).

– Abcisas (eje X): Universo de discurso *X.*
– Ordenadas (eje Y): Grados de pertenecia en el intervalo [0,1].

Ejemplo: Concepto de Temperatura "Alta".

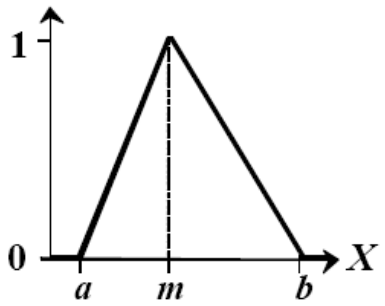


2.1.2. Funciones de Pertenencia

Son dela forma A: X→ [0,1] cuallquier función A es válida: Su definicion exacta depende del concepto a definir, del contexto al que se refiera, de la aplicación.En general, es preferible usar funciones simples, debido a que simplifican muchos cálculos y no pierden exactitud, debido a que precisamente se está definiendo un concepto difuso.

Triangular: Definido por sus límites inferior a y superior b, y el valormodal m, tal que a<m<b.

$$A(x)=\begin{cases}0 & \text{si } x \le a \\ (x-a)/(m-a) & \text{si } x \in (a,m] \\ (b-x)/(b-m) & \text{si } x \in (m,b) \\ 0 & \text{si } x \ge b\end{cases}$$



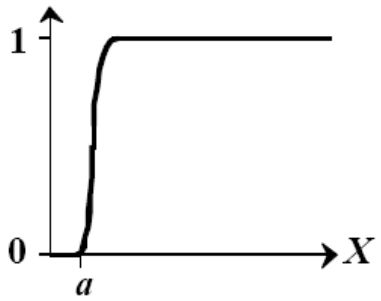
También puede representarse así:

$$A(x;a,m,b) = \max\{\ \min\{\ (x-a)/(m-a),\ (b-x)/(b-m)\ \},\ 0\ \}$$

Función Γ (gamma): Definida por su límite inferior *a* y el valor *k*>0.

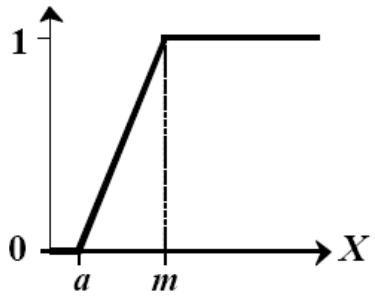
$$A(x)=\begin{cases}0 & \text{si } x \le a \\ 1-e^{-k(x-a)^2} & \text{si } x > a\end{cases}$$

$$A(x)=\begin{cases}0 & \text{si } x \le a \\ \dfrac{k(x-a)^2}{1+k(x-a)^2} & \text{si } x > a\end{cases}$$



- Esta función se caracteriza por un rápido crecimiento a partir de *a*.
- Cuanto mayor es el valor de *k*, el crecimiento es más rápido aún.
- La primera definición tiene un crecimiento más rápido.
- Nunca toman el valor 1, aunque tienen una asíntota horizontal en 1.
- Se aproximan linealmente por:

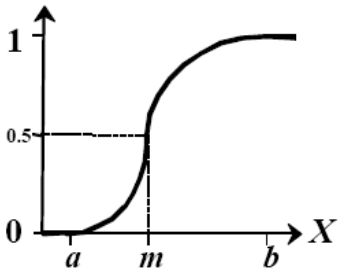
$$A(x) = \begin{cases} 0 & \text{si} \quad x \le a \\ (x-a)/(m-a) & \text{si} \quad x \in (a,m) \\ 1 & \text{si} \quad x \ge m \end{cases}$$



Función S: Definida por sus límites inferior *a* y superior *b*, y el valor *m*, o punto de inflexión tal que *a*<*m*<*b*.

- Un valor típico es: *m*=(*a*+*b*) / *2*.
- El crecimiento es más lento cuanto mayor sea la distancia *a*-*b*.

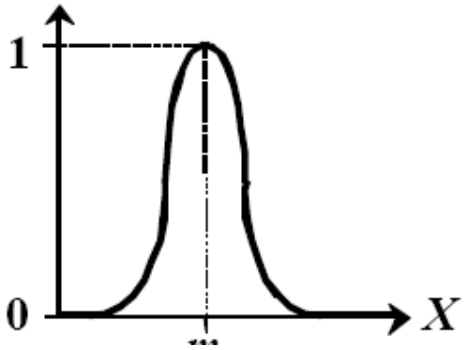
$$A(x) = \begin{cases} 0 & \text{si} \quad x \le a \\ 2\{(x-a)/(b-a)\}^2 & \text{si} \quad x \in (a,m] \\ 1-2\{(x-b)/(b-a)\}^2 & \text{si} \quad x \in (m,b) \\ 1 & \text{si} \quad x \ge b \end{cases}$$



Función Gausiana: Definida por su valor medio m y el valor k>0.

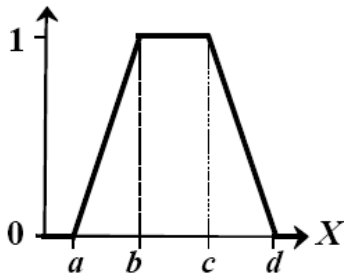
$$A(x) = e^{-k(x-m)^2}$$

- Es la típica campana de Gauss.
- Cuanto mayor es k, más estrecha es la campana.



Función Trapezoidal: Definida por sus límites inferior a y superior d, y los límites de su soporte, b y c, inferior y superior respectivamente.

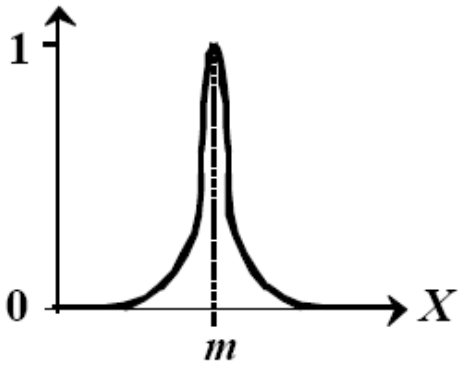
$$A(x)=\begin{cases}0 & \text{si} \quad (x \le a) \ \text{o} \ (x \ge d)\\ (x-a)/(b-a) & \text{si} \quad x \in (a,b]\\ 1 & \text{si} \quad x \in (b,c)\\ (d-x)/(d-c) & \text{si} \quad x \in (b,d)\end{cases}$$



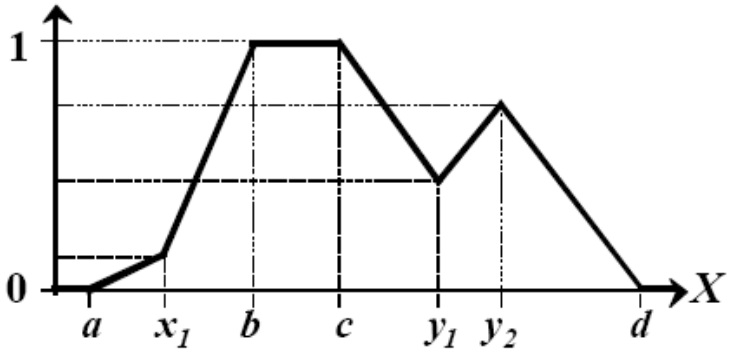
Función Pseudo-Exponencial: Definida por su valor medio m y el valor $k>1$.

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

- Cuanto mayor es el valor de k, el crecimiento es más rápido aún y la "campana" es más estrecha.



Función Trapecio Extendido: Definida por los cuatro valores de un trapecio [a, b, c, d], y una lista de puntos entre a y b, o entre c y d, con su valor de pertenencia asociado a cada uno de esos puntos.



En general, la función Trapezoidal se adapta bastante bien a la definición de cualquier concepto, con la ventaja de su fácil definición, representación y simplicidad de cálculos.

En casos particulares, el Trapecio Extendido puede ser de gran utilidad. Éste permite gran expresividad aumentando su complejidad. En general, usar una función más compleja no añade mayor precisión, pues debemos recordar que se está definiendo un concepto difuso.

2.1.3. Características de un Conjunto Borroso

Altura de un Conjunto Borroso (*height*): El valor más grande de su función de pertenencia: sup $x \in X$ $A(x)$.

Conjunto Borroso Normalizado (*normal*): Si existe algún elemento $x \in X$, tal que pertenece al conjunto borroso totalmente, es decir, con grado 1. O también, que: Altura(A) = 1.

Soporte de un Conjunto Borroso (*support*): Elementos de X que pertenecen a A con grado mayor a 0: Soporte(A) = $\{x \in X \mid A(x) > 0\}$.

Núcleo de un Conjunto Borroso (*core*): Elementos de X que pertenecen al conjunto con grado 1: Nucleo(A) = $\{x \in X \mid A(x) = 1\}$.Lógicamente, Nucleo($A$) $\leq$Soporte(A).

a-Corte: Valores de X con grado mínimo a: $A\alpha$ = $\{x \in X \mid A(x) \geq a\}$.

Cardinalidad de un Conjunto Borroso:con un Universo finito (*cardinality*)**:** Card(A) = S$x \in X$ $A(x)$.

2.1.4. Operaciones Unarias sobre Conjuntos Borrosos

Normalización: Convierte un conjunto borroso no normalizado en uno normalizado, dividido por su altura: Norm_$A(x)$ = $A(x)$ / Altura(A).

Concentración (*concentration*): Su función de pertenencia tomará valores más pequeños, concentrándose en los valores mayores:

- Con_$A(x)$ = $Ap(x)$, con p>1, (normalmente, p=2).
-

Dilatación (*dilation*): Efecto contrario a la concentración. 2 formas:

- Dil_$A(x)$ = $Ap(x)$, con pÎ(0,1), (normalmente, p=0.5).
- Dil_$A(x)$ = 2$A(x)$ – $A2(x)$.

Intensificación del Contraste (*contrast intensification*): Se disminuyen los valores menores a 1/2 y se aumentan los mayores

Difuminación (*fuzzification*): Efecto contrario al anterior:

2.1.5. Relaciones entre Conjuntos Borrosos

Igualdad (*equality*)**:** Dos conjuntos borrosos, definidos en el mismo Universo, son iguales si tienen la misma función de pertenencia:

$$A = B \Leftrightarrow A(x) = B(x), \forall x \in X$$

Inclusión (*inclusion*)**:** Un conjunto borroso está incluido en otro si su función de pertenencia toma valores más pequeños:

$$A \subseteq B \Leftrightarrow A(x) \leq B(x), \forall x \in X$$

Inclusión Borrosa: Si el Universo es finito, podemos relajar la condición anterior para medir el grado en el que un conjunto borroso está incluido en otro (Kosko, 1992):

$$S(A,B) = \frac{1}{\text{Card}(A)} \left\{ \text{Card}(A) - \sum_{x \in X} \max\{0, A(x) - B(x)\} \right\}$$

2.2. Operaciones con conjuntos borrosos

Las operaciones básicas de los conjuntos difusos son:

- **Contención o subconjunto:**
  A es un subconjunto de B si y solo si $\mu_A(x) \leq \mu_B(x)$, para todo x
  $A \subseteq B \Leftrightarrow \mu_A(x) \leq \mu_B(x)$

- **Unión:**
  La unión de los conjuntos difusos A y B es el conjunto difuso C, y se escribe como C= AOR B, su función de dependencia está dada por
  $\mu_C(x) = \max(\mu_A(x), \mu_B(x)) = \mu_A(x) - \mu_B(x)$

- **Intersección:**
  La intersección de los conjuntos difusos A y B es el conjunto difuso C, y se escribe como C=A AND B, su función de dependencia está dada por
  $\mu_C(x) = \min(\mu_A(x), \mu_B(x)) = \mu_A(x) \cdot \mu_B(x)$

- **Complemento (negación):**
  El complemento del conjunto difuso A, denotado por $\bar{A}$ ($\neg$A, NOT A), se define como $\mu_{\bar{A}}(x) = 1 - \mu_A(x)$

- **Producto Cartesiano:**

  Si A y B son conjuntos difusos en X e Y, el producto cartesiano de los conjuntos A y B A x B en el espacio de X x Y tiene la función de pertenencia

  $\mu_{AxB}(x,y) = \min(\mu_A(x), \mu_B(x))$

- **Co-producto Cartesiano**

  A + B en el espacio X x Y tiene la función de pertenencia $\mu_{AxB}(x,y) = \max(\mu_A(x), \mu_B(x))$

2.2.1. Propiedades Básicas

Conmutativa: $A \cup B = B \cup A$; $A \cap B = B \cap A$;

Asociativa: $A \cup (B \cup C) = (A \cup B) \cup C = A \cup B \cup C$;

$A \cap (B \cap C) = (A \cap B) \cap C = A \cap B \cap C$;

Idempotencia: $A \cup A = A$; $A \cap A = A$;

Distributiva: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$;

$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$;

Condiciones Frontera o Límite: $A \cup \phi = A$; $A \cup X = X$; $A \cap \phi = \phi$; $A \cap X = A$;

Involución (doble negación): $\neg(\neg A) = A$;

Transitiva: $A \subset B$ y $B \subset C$, implica $A \subset C$;

Propiedades Añadidas: Se deducen de las anteriores.

$(A \cap B) \subset A \subset (A \cup B)$;

Si $A \subset B$, entonces $A = A \cap B$ y $B = A \cup B$;

$Card(A) + Card(B) = Card(A \cup B) + Card(A \cap B)$;

$Card(A) + Card(\neg A) = Card(X)$;

2.2.2. Normas y Conormas Triangulares

Son conceptos derivados de Menger (1942) y Schwizer y Sklar (1983), actualmente están muy desarrollados (Butnario et al., 1993).Establecen modelos genéricos para las operaciones de unión y intersección, las cuales deben cumplir ciertas propiedades básicas (conmutativa, asociativa, monotonicidad y condiciones frontera).

Norma Triangular, t-norma: Operación binaria t: $[0,1]^2 \rightarrow [0,1]$ que cumple las siguientes propiedades:

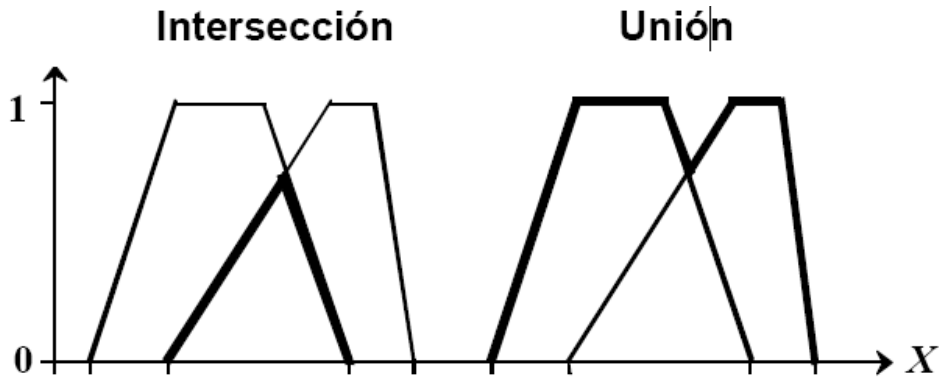
- **Conmutativa**: $x \text{ t } y = y \text{ t } x$
- **Asociativa**: $x \text{ t } (y \text{ t } z) = (x \text{ t } y) \text{ t } z$
- **Monotonicidad**: Si $x \leq y$, y $w \leq z$ entonces $x \text{ t } w \leq y \text{ t } z$
- **Condiciones Frontera**: $x \text{ t } 0 = 0$, $x \text{ t } 1 = x$

Conorma Triangular, t-conorma o **s-norma:** Op. bin. s: $[0,1]^2 \rightarrow [0,1]$ que cumple las siguientes propiedades:

- **Conmutativa:** $x \text{ s } y = y \text{ s } x$
- **Asociativa:** $x \text{ s } (y \text{ s } z) = (x \text{ s } y) \text{ s } z$
- **Monotonicidad:** Si $x \leq y$, y $w \leq z$ entonces $x \text{ s } w \leq y \text{ s } z$
- **Condiciones Frontera:** $x \text{ s } 0 = x$, $x \text{ s } 1 = 1$

t-norma del mínimo: La función mín es una t-norma, que corresponde a la operación de intersección en conjuntos clásicos cuyos grados de pertenencia están en {0,1}. Por eso, esta función es la extensión natural de la intersección en conjuntos difusos.

t-conorma o s-norma del máximo: La función máx es una s-norma, que corresponde a la operación de unión en conjuntos clásicos cuyos grados de pertenencia están en {0,1}. Por eso, esta función es la extensión natural de la unión en conjuntos difusos.



2.2.3. Operaciones de agregación

Son operaciones que combinan una colección de conjuntos difusos para producer un único conjunto difuso.

Una Agregación es una operación *n*-aria A: $[0,1]^n \rightarrow [0,1]$, que cumple:

Condiciones Frontera: A(0,...,0) =0 y A(1,...,1) = 1
Monotonicidad: A(x1,...,xn) A(y1,...,yn) si xi. yi, i=1,...,n

Las t-normas y s-normas ($\cap$ / U) son operaciones de agregación.

- **Operadores Compensatorios:** A veces, una t-norma/s-norma no se comporta demasiado bien modelando las operaciones de ∩ / U (conectivos **AND / OR**). Zimmermann y Zysno **[Zimmermann y Zysno 1980]** propusieron el siguiente operador compensatorio Θ (*zeta*), donde el factor de compensación $\gamma \in [0,1]$indica en qué punto está situado el operador entre AND y OR.

$$(A\Theta B)(x) = (A \cap B)(x)^{1-\gamma} (A \cup B)(x)^{\gamma}$$

Cuanto mayor es g más importancia tiene la U respecto a la ∩.

- **Sumas Simétricas:**Son funciones de *n* argumentos que además de cumplir la monotonicidad y las condiciones frontera, son continuas, conmutativas y auto-duales: *S*_sum (*x*1, ... , *xn*) = *S*_sum (1–*x*1, ... , 1–*xn*).

$$S\_\text{sum}(x_1,\ldots,x_n) = \left[1 + \frac{\rho(1-x_1,\ldots,1-x_n)}{\rho(x_1,\ldots,x_n)}\right]^{-1}$$

- **Operadores OWA** (*Ordered Weighted Averaging*)[**Yager, 1988**]:Son operadores ponderados con un vector de pesos *wi* tal que si ordenamos los valores {*A*(*xi*)}: *A*(*x*1) ≤ *A*(*x*2) ≤ . . . ≤ *A*(*xn*) queda:

  • OWA(*A*, (1,0,...,0)) = mín{*A*(*x*1), *A*(*x*2), . . . , *A*(*xn*)}.
  • OWA(*A*, (0,...,0,1)) = máx{*A*(*x*1), *A*(*x*2), . . . , *A*(*xn*)}.
  • OWA(*A*, (1/*n*, ..., 1/*n*)) = (1 / *n*) $\sum_{i=1}^{n}$ *A*(*xi*)

Operación Media: Funciones de n argumentos que cumple las propiedades: idempotencia, conmutativa y monotonicidad.

2.2.4. Negaciones de conjuntos borrosos

Complemento o Negación de un conjunto difuso: N: [0,1]→[0,1] cumpliendo las siguientes condiciones:

- Monotonicidad: N es no creciente.
- Condiciones Frontera: $N(0)=1$, $N(1)=0$;

Pueden añadirse otras propiedades, si es necesario:

- Continuidad: N es una función continua.
- Involución: $N(N(x)) = x$, para $x \in [0,1]$;

2.3. Variables lingüísticas

La Variable Lingüística (*Linguistic Variable*) es una variable cuyos valores son palabras o sentencias (no números). A menudo queremos describir el estado de un objeto o fenómeno y para ello usamos una variable cuyo valor hace la descripción.

Ejemplos: Temperatura, Limpieza, Sabiduría...

Una variable lingüística admite que sus valores sean Etiquetas Lingüísticas, que son términos lingüísticos definidos como conjuntos borrosos (sobre cierto dominio subyacente).

Ejemplos: Temperatura “Cálida”, o “aproximadamente 25ºC” (#25). El dominio subyacente es un dominio numérico: Los grados centígrados.

Un valor concreto, *crisp* (25ºC, por ejemplo), Es, en general, más específico que una etiqueta lingüística. Es un punto del conjunto, mientras que una etiqueta lingüística es una colección de puntos (temperaturas posibles).

Hay variables cuya definición es más compleja porque se mueven en dominios subyacentes poco claros y no es natural trasladarlos a valores numéricos: Limpieza, Sabiduría, Verdor...

El uso de variables es una forma de comprimir información **[Zadeh 1994a,b]** llamada granulación (*granulation*)**.** Una etiqueta incluye muchos valores posibles.

Ayuda a caracterizar fenómenos que o están mal definidos o son complejos de definir o ambas cosas **[Zadeh 1975].**

Es un medio de trasladar conceptos o descripciones lingüísticas a descripciones numéricas que pueden ser tratadas automáticamente:Relaciona o traduce el proceso simbólico a proceso numérico.

Usando el principio de extensión, muchas herramientas ya existentes pueden ser extendidas para manejar variables lingüísticas, obteniendo las ventajas de la lógica borrosa en gran cantidad de aplicaciones.

Definición formal de Variable Lingüística **[Zadeh, 1975]:** Es un conjunto de 5 elementos**: <N, U, T(N), G, M>**

- **N** es el nombre de la variable y **U** dominio subyacente.
- **T(N)** es el conjunto de términos o etiquetas que puede tomar **N**.
- **G** es una gramática para generar las etiquetas de **T(N)**: “muy alto”, “no muy bajo”, “extremadamente normal”, “bajo y normal”...
- **M** es una regla semántica que asocia cada elemento de **T(N)** con un conjunto borroso en **U** de entre todos los posibles**: M: T(N) → F (U)**

Gramática G: Normalmente los símbolos terminales incluyen:

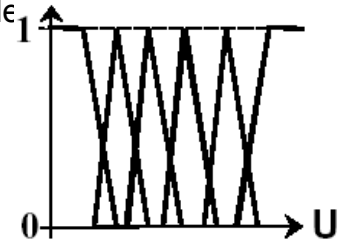
- Un conjunto de términos primarios (*primary terms*): Bajo, Alto...
- Un conjunto de modificadores (*hedges*): Muy, más o menos, completamente, especialmente, más, cerca de...
- Un conjunto de conectivos lógicos: Normalmente NOT, AND y OR.

Las funciones de pertenencia suelen ser de uno de los tipos clasificados ya referidos en el capítulos anteriores**:**

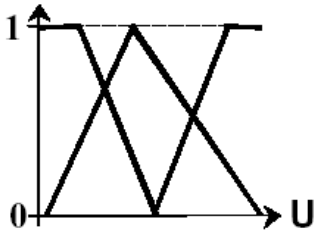
- Triangulares, Trapezoidales, Gamma...

Normalmente se usa un conjunto pequeño de valores para una variable lingüística lo que define su granularidad, pudiéndo ser esta de:

- **Granularidad Fina (*fine*):** Define un gran número de valores para una variable



- **Granularidad Gorda (*coarse*):** Define un pequeño número de valores.



2.3.1. Modificadores lingüísticos

Una Etiqueta Lingüística se forma como una sucesión de los símbolos terminales de la gramática: Muy Alto, No Muy Bajo...

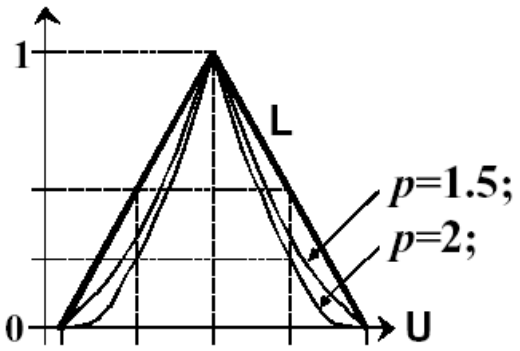
Normalmente se definen los conjuntos borrosos de los términos primarios y, a partir de éstos se calculan los conjuntos borrosos de los términos compuestos **[Zadeh, 1972]**

Cada modificador (*hedge*) es un operador H que transforma el conjunto borroso del término primario L al que afecta en otro conjunto borroso:

Básicamente, se usan las operaciones siguientes **[MacVickar-Whelan, 1978; Zadeh, 1975]:**

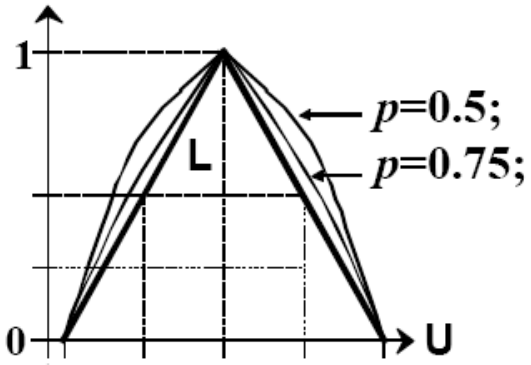
Concentración: Elevar a *p* con *p*>1:

"Muy L" o "Muy aproximadamente igual a L" (*p*=2),"Más L" (*p*=1.5)...



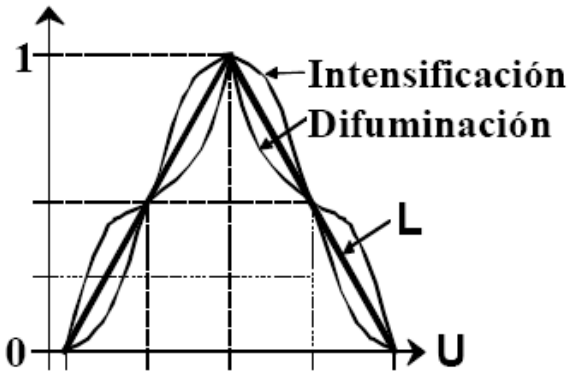
Dilatación: Raíz *n*-ésima o elevar a $p \in (0,1)$:

"Más o menos L" (*p*=0.5),"Menos L", "Poco L" (*p*=0.75)...



Intensificación del contraste: Disminuir valores menores que 0.5 y aumentar los otros: "Especialmente L","Bastante cerca de L"...

Difuminación: Efecto contrario: "Cerca de L","Casi L"...



Los **Operadores Lógicos** son evaluados como operadores de conjuntos borrosos**:**

NOT: Complemento o negación.
AND: Intersección (t-norma).
OR: Unión (s-norma).

Pueden construirse nuevos modificadores lingüísticos a partir de las expresiones anteriores. “Ligeramente L” puede traducirse por “L y no muy L”. L debe estar normalizado para que el resultado sea coherente. Incluso pueden definirse otras expresiones: “Entre L y M”, “Mucho mayor que L”,“Mucho menor que L”, “A partir de L·, “Mayor o igual que L”...

2.3.2. Cuantificadores Lingüísticos

Los cuantificadores lingüísticos se usan para medir (o cuantificar) la cantidad o la proporción de objetos o elementos que cumplen o satisfacen cierta condición.

En lógica clásica existen dos muy importantes:

• " (todo): Se refiere a todos los elementos u objetos.
• $ (existe): Se refiere al menos a uno de los elementos u objetos.

Usando lógica borrosa existen más clasificados en dos categorías:

Cuantificadores Absolutos: Se refieren a una única cantidad determinada para medir si esa cantidad son “muchos”, “pocos”,“muchísimos”, “aproximadamente entre 6 y 9”, “aprox. más de 43”,“aprox. 8”... Para evaluar la verdad de un cuantificador absoluto necesitamos una única cantidad.

Cuantificadores Relativos: Se refieren a una proporción de elementos respecto del total de los que existen. Por ejemplo:“la mayoría”, “la minoría”, “casi todos”, “casi ninguno”,“aprox. la mitad”... Para evaluar la verdad

necesitamos 2 cantidades: Los elementos que cumplen la condición y el total de elementos existentes.

Los Cuantificadores Borrosos se representan como conjuntos borrosos con dominio subyacente en los números reales [**Zadeh, 1983].** Este Dominio está limitado dependiendo del tipo de cuantificador **[Kacprzyk, Fredizzi, Nurmi, 1992; Yager, 1983; Zadeh, 1983]:**

Cuantificadores Borrosos Absolutos: *Q*abs: R+ → [0,1]

Cuantificadores Borrosos Relativos : *Q*rel: [0,1] → [0,1]

En los relativos, el cuantificador se aplica a la división del número de elementos que cumplen la condición entre el número de elementos totales. Las sentencias cuantificadas pueden ser de dos formas:

***Q x*'s son *B* → Ej.:** "La mayoría de los alumnos son jóvenes"

El grado de verdad de este tipo de sentencias se evalúa como:

***Q*(*r*)**, donde ***r* = Card(*B*) / Card(*X*) = (å *i* B(*xi*))/*n*;**

Siendo ***X*** el universo de discurso de ***n*** elementos.

***Q A x*'s son *B*→ Ej.:** "La mayoría de los alumnos altos son jóvenes"

Grado de verdad: ***Q*(*r*)**, donde ***r* = Card(*A*|*B*) / Card(*A*);**

2.4. Lógica borrosa y sistemas basados en reglas

La Lógica Borrosa (Fuzzy Logic) es una generalización de la lógica multivaluada. Permite utilizar conceptos "aproximados", por lo que el razonamiento también será "aproximado". En Lógica Borrosa todo es cuestión de GRADO, incluso la verdad **[Zadeh, 1975 y 1988**]: Un Grado de Verdad puede ser: un Valor Numérico del intervalo [0,1]. Ejemplos: 0.5,0.75... o una

Etiqueta Lingüística. Ejemplos: más o menos verdad, bastante... Resumiendo, un grado de verdad es un conjunto borroso.

Con estas bases, han surgido trabajos en diversas líneas, una de estas líneas, y la que más nos interesa, es la ver la lógica borrosa como una herramienta para a resolución de problemas y la toma de decisiones **[Zadeh, 1975 y 1979;Tsukamoto, 1979; Pedrycz, 1995].** Para ello utilizan la inferencia lógica aplicada a los conjuntos borrosos de los grados de verdad.

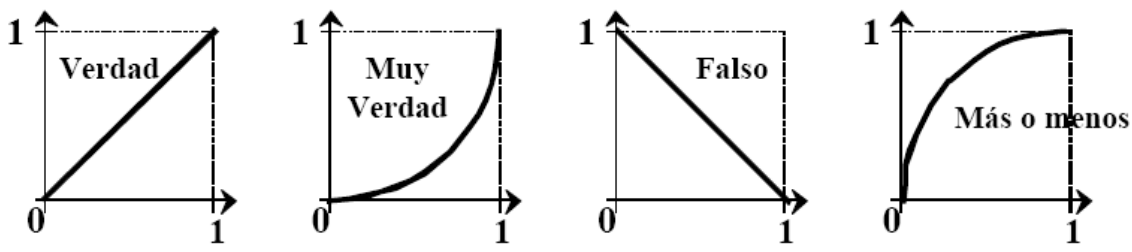
2.4.1. Cálculos con Lógica Borrosa

La cualificación de verdad (Truth Qualification)**[Zadeh, 1975]** consiste en obtener un conjunto difuso A tal que: "X es Ai" es t_i = "X es A"

donde t_i actúa como una restricción elástica: $A(x) = t_i(Ai(x)), \quad \forall x \in X$

$A(x)$ = Verdad($Ai(x)$) = $Ai(x)$;
$A(x)$ = Muy Verdad($Ai(x)$) = $A^2 i(x)$;
$A(x)$ = Falso($Ai(x)$) = $1 - Ai(x)$;
$A(x)$ = Más o Menos($Ai(x)$)=$A^{0.5} i(x)$;

Si t_i=Falso, se está afirmando el hecho contrario. Por eso, podemos definir t_i=Totalmente Falso que toma el grado 0 en todo su universo [0,1], excepto para el valor 0, que toma grado 1.



La cualificación de verdad inversa por otro lado consiste en obtener el conjunto difuso t_i partiendo de los conjuntos A y Ai. La fórmula se basa en el principio de extensión:

$$\tau_i(v) = \sup_{x : A_i(x)=v} \{A(x)\};$$

Si tenemos dos proposiciones con dos grados de verdad t_A y t_B, deducimos que:

AND Difusa:

$$\tau_{A\wedge B}(v) = \sup_{w,z\in[0,1] v=w\,\mathbf{t}\,z} \{\tau_A(w)\wedge\tau_B(z)\};$$

OR Difusa:

$$\tau_{A\vee B}(v) = \sup_{w,z\in[0,1] v=w\,\mathbf{s}\,z} \{\tau_A(w)\wedge\tau_B(z)\};$$

NOT Difuso:

$$\tau_{\neg A}(v) = \sup_{u\in[0,1] v=1-u} \{\tau_A(u)\} = \tau_A(1-u);$$

Implicación Difusa:

$$\tau_{A\to B}(v) = \sup_{w,z\in[0,1] v=w\to z} \{\tau_A(w)\wedge\tau_B(z)\};$$

2.4.2. Razonamiento o Inferencia

Partiendo de la fórmula general { A, $A_i \rightarrow B_i$ [es t_i] } $\Rightarrow$ B donde $A_i \rightarrow B_i$ es una regla que se cumple en el sistema (implicación) con el grado de verdad t_i (opcional), A es el dato de entrada o situación actual y B es la conclusión: Si A=Ai, entonces B=Bi.

Siendo t*i* un valor de verdad lingüístico, la inferencia consta de 3 fases:

1. **Cualificación de Verdad Inversa**:Obtener t_{Ai} como la compatibilidad de A_i respecto a A.

2. **Inferencia Lógica Difusa:** Usando la implicación difusa a partir del grado de verdad de la regla y del antecedente: t_{Bi}.

3. **Cualificación de Verdad:** Obtener B con B_i y el grado de inferencia t_{Bi}.

2.4.3. Sistemas Basados en Reglas

Las Reglas son un modo de representar estrategias o técnicas apropiadas cuando el conocimiento proviene de la experiencia o de la intuición (careciendo de demostración matemática o física).

Su formato son Proposiciones que usan IF–THEN (SI–ENTONCES):

IF <antecedente o condición> THEN<consecuente o conclusión>

El <antecedente> y el <consecuente> son proposiciones borrosas que pueden formarse usando conjunciones (AND) o disyunciones (OR): El significado, obviamente, depende de esto.

Ejemplo: SI la Temperatura es Alta ENTONCES Abrir la válvula Poco.

Para realizar la generalización de reglas primero se identifican las variables que intervienen (Temperatura, Nivel de apertura de la válvula...) y sus valores posibles: Es normal que en las reglas se representen más los valores que las variables (Ej.: Si hace calor...).Después se Identifican las restricciones que inducen las proposiciones y por último se representan cada restricción con una relación borrosa (regla).

Existen diferentes tipos de proposiciones están las proposiciones cualificadas y las proposiciones cuantificadas:

- Las proposiciones cualificadas(Qualified Propositions) añaden un grado (o etiqueta lingüística) a la proposición que forma una regla, estos grados pueden ser de distinto tipo, por ejemplo: grados de certeza (verdad, falso, casi verdad...), grados de probabilidad (Probable, poco probable, normalmente...) o grados de posibilidad (Posible, Poco Posible...).
- En las proposiciones cuantificadas(Quantified Propositions) pueden usarse cuantificadores borrosos: Muchos, Pocos, la Mayoría, Frecuentemente, Aproximadamente 8...

Ejemplos:

La Mayoría de los Alumnos Listos son Ordenados.

Frecuentemente, SI la Temperatura es Alta, ENTONCES la Válvula está Poco Abierta.

También se incluyen en esta categoría las reglas cuantificadas en el antecedente (Antecedent-Quantified):Si se pone un cuantificador en el antecedente.

Ejemplo:

SI se cumplen LA MITAD de las condiciones, ENTONCES...

2.4.4. Sintaxis de las Reglas Borrosas

Existen distintos formatos posibles de la sintaxis de las reglas difusas:

– El <Atributo> del <Objeto> es <Valor> → La Humedad del Suelo es Alta

<Atributo> es una Variable Lingüística o Atributo del <Objeto>.

<Valor> es una Etiqueta Lingüística de ese Atributo.

– <Atributo> (<Objeto>) es <Valor> → Humedad(Suelo) es Alta.

– <AtributoDeUnObjeto> es <Valor> →Humedad es Alta.

Es el formato más usual representado como: X es A.

En el caso de las **proposiciones cualificadas** de la forma: *X* es *A* con certeza $\mu \in$ [0,1]. Pueden transformarse en proposiciones con certeza 1: *X* es *B* donde *B* es calculada por [Yager, 1984]:

$$B(x) = [\mu \text{ t } A(x)] + (1 - u)$$

Si μ=1, entonces *B*=*A*.

Si μ=0, entonces *B*=*U* (Universo de *X*).

El valor *B* tiene menos especificidad que el original *A*.

Cuanto mayor es la certeza μ, mayor será la especificidad de *B*.

Las **proposiciones compuestas** usan conjunciones o disyunciones. Esta forma induce relaciones difusas (P) sobre las variables (Xi), definidas con una t-norma T o una s-norma S, sobre las etiquetas lingüísticas (Ai), (según sean conjunciones o disyunciones respectivamente.):

Conjunciones:

$$P(x_1,...x_n) = \prod_{i=1}^{n} A_i(x_i);$$

Disyunciones:

$$P(x_1,...x_n) = \sum_{i=1}^{n} A_i(x_i);$$

Estas proposiciones pueden ser expresadas también como: (X_1,...,X_n) es P;

Las **reglas simples** son de la forma:

Si X es A, entonces Y es B

Puede ponerse como "(X, Y) es P", donde P es una relación difusa definida en los universos de X e Y: P: X x Y →[0,1]

Las **reglas con proposiciones compuestas** puede ponerse como "(X1, ..., Xn, Y1, ..., Ym) es P", donde P es una relación difusa definida en los universos de las variables del antecedente (Xi) y del consecuente (Yi):

$P(x1 ,. ..xn , y1 ,... ym,) = f (Pa (x1 ,... xn), Pc (y1 ,... ym))$;

donde f es un operador de Implicación o una t-norma y, Pa y Pc son relaciones inducidas por el antecedente y el consecuente respec.

En el caso de las **reglas cualificadas** de la forma:

Si ... Entonces ... con certeza m.

Si su forma equivalente usa la relación P: "(X1, ..., Xn, Y1, ..., Ym) es P con certeza m", puede usarse la relación Q con certeza 1:

$$Q(x1, ..., xn, y1, ..., ym) = [m \text{ t } P(x1, ..., xn, y1, ..., ym)] + (1 - m).$$

2.4.5. Cálculos de las reglas difusas

Para calcular los **antecedentes compuestos:**

Tengamos una colección de *N* reglas del tipo: *k* = 1, 2, ... , *N* "Si *X* es *Ak* y *Y* es *Bk*, Entonces *Z* es *Ck*"

En ese caso, se toma como si el antecedente fuera del tipo:

"(*X*, *Y*) es *Pk*", donde *Pi* es calculada con una t-norma: *Pk* (*x*, *y*) = *Ak* (*x*) t *Bk* (*y*) Si el operador fuera la disyunción (o), se tomaría una s-norma.

Entradas *crisp* para *X* e *Y* : *a* y *b* respectivamente. Con entradas *crisp(nítidas)* los cálculos se simplifican mucho.

Sea *mk* el valor resultante de aplicar la t-norma a los valores obtenidos en el antecedente de la Regla *k*: *mk* = *Ak* (*a*) t *Bk* (*b*). *mk* es llamado "Grado de Activación" (*Activation Degree*) y mide la contribución de la regla *k* en la inferencia global.

– El conjunto difuso resultante *C* es calculado como la unión de los conjuntos difusos *C'k* obtenidos en cada regla:

$$C(z) = \bigcup_{k=1}^{N} C'_k = \mathbf{S}_{k=1}^{N}(m_k \ \mathbf{t} \ C_k(z)), \qquad \forall z \in \mathbf{Z};$$

Al efectuar una inferencia sobre un conjunto de reglas, debemos elegir apropiadamente:

– Una t-norma para definir el operador de conjunción (y) y una snorma para el operador de disyunción (o), que se aplicará en el antecedente y el consecuente de cada regla.
– Una función *f* para definir el significado de cada regla *k*, o sea el significado de la Implicación (t-norma usada en el cálculo de *F**).
– Una t-norma para la Regla Composicional de Inferencia.
– Un operador de Agregación Ag para la Regla de ombinación (s-norma utilizada en el cálculo de *F**).

Si se disparan N reglas simples del tipo “Si X es Ai, entonces Y es Bi”, sabiendo que el valor de la variable de entrada X es A, el valor de la variable de salida Y será el conjunto difuso:

$$B(y) = \sup_x \left(A(x) \ \mathbf{t} \ \mathop{\mathbf{Ag}}_{k=1}^{N} \left(f\left(A_k(x), B_k(y)\right)\right)\right) = \mathop{\mathbf{Ag}}_{k=1}^{N} \left(\sup_x \left(A(x) \ \mathbf{t} \ f\left(A_k(x), B_k(y)\right)\right)\right);$$

La Regla Composicional de Inferencia puede aplicarse también localmente a cada regla y agregar los resultados al final.

Si se disparan N Reglas compuestas usando operadores de conjunción (y) en el antecedente y el consecuente: k = 1, 2, ... , N “Si X1 es A1k y X2 es A2k y ... y Xn es Ank Entonces Y1 es B1k y Y2 es B2k y ... y Ym es Bmk”

Siendo los datos de entrada: *X*1 es *A*1 y *X*2 es *A*2 y ... y *Xn* es *An,* entonces el Resultado sería:

$$B(y1, y2, \ldots, y = \mathop{\mathbf{Ag}}_{k=1}^{N} \left(\sup_x \left(A(x_1, x_2, \cdots, x_n) \ \mathbf{t} \ f\left(A_k(x_1, x_2, \cdots, x_n), B_k(y_1, y_2, \cdots, y_m)\right)\right)\right);$$

Donde A se obtiene de aplicar la t-norma a las entradas, A_k se obtine de aplicar la t-norma en el antecedente y Bk se obtiene de aplicar la t-norma en el consecuente. En el caso de ser con el operador de disyunción (o) se aplicaría una s-norma.

El proceso general es el siguiente:

1. Emparejar Antecedentes y Entradas:

• Para cada REGLA se calcula el grado de emparejamiento entre cada proposición atómica de su antecedente y el valor correspondiente de la entrada.

2. Grado de Activación o Agregación de los Antecedentes:

• Para cada REGLA se calcula el Grado de Activación aplicando una conjunción (t) o disyunción (s) según corresponda a los valores anteriores del Paso 1.

3. Resultado de cada Regla:

• Para cada REGLA se calcula su valor resultante según su Grado de Activación y la semántica elegida para la Regla. Este es el paso más largo y complejo: Para cada valor en las Salidas se debe calcular el mayor valor de la operación, para todos los posibles valores de las Entradas (operación supx).

4. Regla de Combinación:

• Agregación de todos los resultados individuales obtenidos de cada una de las reglas aplicadas.

2.5. Modelos borrosos

Los modelos borrosos son una aplicación muy útil de los conjuntos borrosos **[Bezdek,1993; Pedrycz, 1993a, 1995; Zadeh, 1965].** Su objetivo es construir un modelo para un determinado sistema con las siguientes características:

- Operar a nivel de términos lingüísticos (conjuntos borrosos).
- Representar y procesar incertidumbre, imprecisión o vaguedad de los datos. **[FORT,2005]**

Un modelo borroso tendrá un comportamiento determinado dependiendo de la granularidad de los conjuntos borrosos definidos, por ello, es importante que el diccionario(que lo conforman la base de conocimiento junto con la base de reglas) utilizado por el modelo sea elegido cuidadosamente.

2.5.1. Fases del Modelado de Sistemas

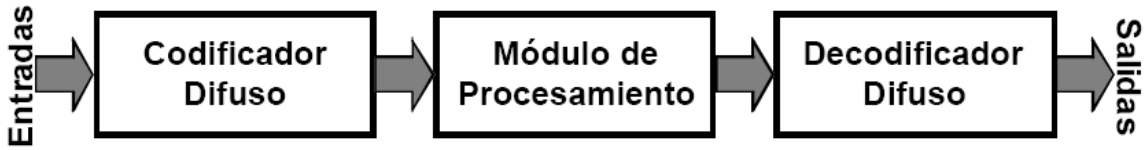
El desarrollo de un modelo tiene las siguientes fases principales:

1. **Preprocesamiento**: Especificación de las variables de entrada y de salida y el estudio del conocimiento relevante.
2. **Estimación de Parámetros**: Se eligen los parámetros del sistema usando alguna técnica de optimización.
3. **Verificación del Modelo:** Se verifica su funcionamiento según los datos disponibles y se cuantifica el error producido (por ejemplo, mediante la suma del cuadrado de los errores).
4. **Validación del Modelo:** Se trata de asegurar que el modelo es válido, soluciona los problemas planteados y se comporta como el usuario esperaba.

Las dos últimas fases suelen llamarse proceso VV (Verificación y Validación del modelo).

2.5.2. Topología del modelado borroso

Una arquitectura general de un modelo borroso es:



Los conjuntos borrosos forman la interfaz entre el módulo de procesamiento y el entorno de una aplicación particular. Esto nos permite ver el entorno desde la perspectiva más relevante, si escogemos un nivel de granularidad apropiado **[Zadeh, 1979;Pedrycz, 1992]** y además se preprocesan los datos antes (y después) de que el módulo de procesamiento los use. Ese preprocesamiento cambia si cambiamos la forma de los conjuntos borrosos o el número de ellos definidos.

Codificación/Decodificación: Ambos mecanismos deben ser compatibles. Se trata de intentar conseguir un canal de comunicación sin pérdidas:

decodificar (codificar(X)) = X

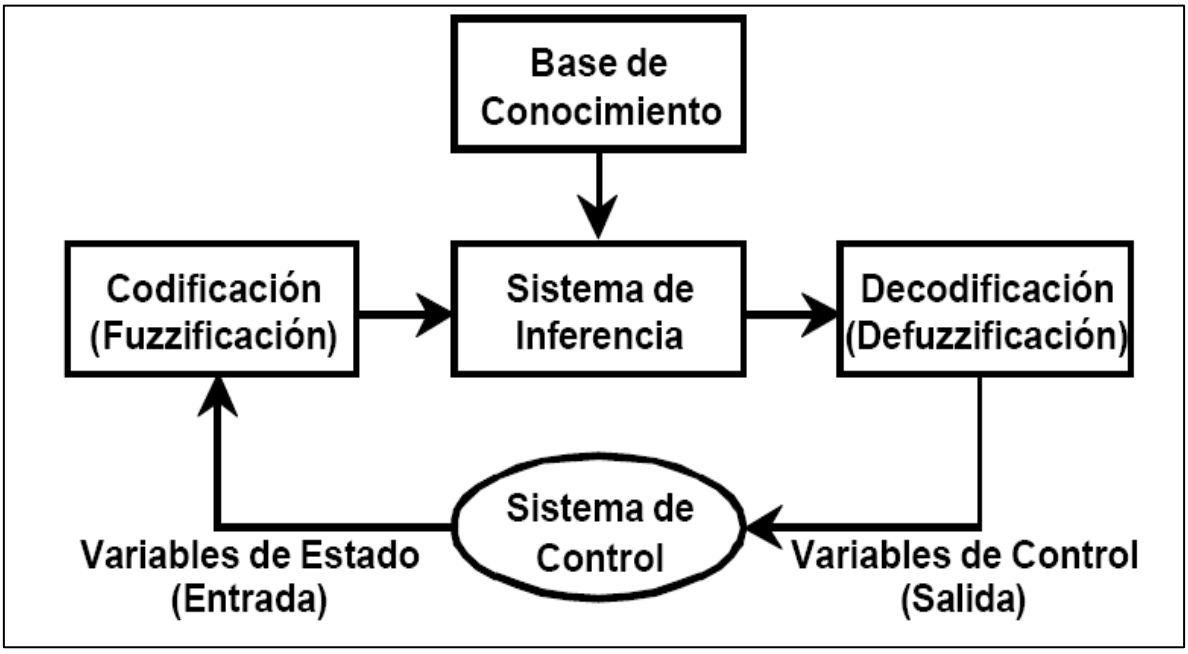
donde X es un fragmento de la información de entrada.

Módulo de Procesamiento: Es la parte principal del modelo, donde se almacena el comportamiento del mismo.Puede ser visto como un conjunto de reglas encapsuladas en redes neuronales borrosas (Pedrycz, 1993b) o en un S.B.R. Borrosas y en una función discriminante (lineal o no lineal): En este caso estaremos interesados en construir clasificadores lingüísticos.Da lugar a diversas clases de modelos borrosos.

2.6. Control borroso

La estructura genérica de un **Controlador Difuso** tiene 4 componentes principales**[Lee, 1990]:**

- **Base de Conocimiento.**
- **Sistema de Codificación.**
- **Sistema de Inferencia.**
- **Sistema de Decodificación.**



Base de Conocimiento

La base de conocimiento contiene el conocimiento asociado al dominio de la aplicación y los objetivos del control. Está formada por una base de datos y un conjunto de reglas difusas de control de la forma:

SI (*X*1 es *A*1) y (*X*2 es *A*2) y ... (*Xn* es *An*) **ENTONCES** (*Y* es *B*)

Donde los *Xi* son las variables de estado del sistema a controlar. *Y* es una variable de control del sistema a controlar y los *Ai* y *B* son etiquetas lingüísticas con una función de pertenencia asociada que dan valor a sus variables.

Los dos objetivos principales que cumple este módulo son los de proporcionar las definiciones necesarias para determinar las reglas lingüísticas de control y de manipulación de los datos difusos del controlador y almacenar los objetivos y la política de control (como un experto en el dominio).

Sistema de Fuzzificación

El sistema de fuzzificación realiza el escalado de los valores de las entradas para adecuarlos a los valores típicos para los que se define el sistema, y una "borrosificación" que convierte los datos de entrada en valores lingüísticos adecuados para la manipulación de éstos como entidades borrosas.

Sistema de Inferencia

El sistema de inferencia es el núcleo del controlador difuso y agrupa toda la lógica de inferencia borrosa del sistema, de barrido de las reglas durante ésta, la elección refinada de las reglas a utilizar, etc. El sistema de inferencia infiere las acciones de control usando una implicación difusa y las reglas de inferencia de la lógica difusa.

Sistema de Defuzzificación

El sistema de defuzzificación convierte los valores difusos de las variables de salida en valores concretos dentro del universo de discurso correspondiente. Genera una acción no difusa a partir de la acción difusa resultante del sistema de inferencia. Existen diversos métodos, explicados en capítulos anteriores.

Un tipo de controlador interesante son los adaptativos que son controladores que se reajustan automáticamente para adaptarse a nuevas características del proceso a controlar. Éstos tienen como comonentes:

– **Monitor del Proceso:** Encargado de detectar los cambios en las características del proceso. Puede ser:

- Una medida del rendimiento que refleja la bondad de la actuación del controlador.
- Un parámetro basado en el estado del proceso.

– **Mecanismo de Adaptación**: Utiliza la información del Monitor del Proceso para actualizar los Parámetros del Controlador:

- Factor de escala de cada variable.
- Conjunto difuso de cada etiqueta lingüística.
- Reglas de la Base de Conocimiento.

Esos 3 parámetros son invariantes en un controlador no adaptativo, en el caso de los autoorganizativos éstos pueden partir sin reglas para aprenderlas.

2.6.1. Características de los Controladores Difusos [Sur, Omron, 1997]

– **Son bastante Intuitivos**: La posibilidad de usar expresiones con imprecisión genera modelos intuitivos.

– **Tolerancia al Ruido**: En general, como una salida depende de varias reglas no se verá muy afectada si se produce una perturbación (ruido).

– **Estabilidad:** Son sistemas robustos. En caso de caída del sistema ésta se produce lentamente, dando tiempo a tomar medidas y pueden alcanzar rápidamente la estabilidad en etapas transitorias.

– **No necesita un modelo matemático** preciso del sistema a controlar. Permiten controlar sistemas que son imposibles de controlar con los sistemas de control clásicos.

– **Permiten gran Precisión**: Similar a los sistemas no difusos.

3. ESTUDIO Y RESOLUCIÓN DEL PROBLEMA

3.1. Modelo propuesto

En apartados anteriores hicimos mención a las ventajas que la orientación a objetos puede proporcionar nuestro diseño, a continuación detallamos dichas ventajas:

Uniformidad: Ya que la representación de los objetos lleva implícito tanto el análisis como el diseño y la codificación de los mismos. El modelo orientado a objetos facilita la integridad de módulos que hallan sido realizados por separado sin correr riesgos en el manejo de los datos. Además un modelo de objetos es más cercano a la realidad que un modelo funcional.

Comprehensión: Los datos que componen los objetos, como los procedimientos que los manipulan están agrupados en clases, que se corresponden a las estructuras de información que el programa trata. El modelo orientado a objetos evita la redundancia en los procesos luego, los códigos son más entendibles y resumidos.

Flexibilidad: En las relaciones entre los procedimientos que manipulan los datos, cualquier cambio se ve reflejado automáticamente en cualquier lugar donde estos datos aparezcan.

Estabilidad: Dado que permite un tratamiento diferenciado de aquellos objetos que permanecen constantes en el tiempo, sobre aquellos que cambian con frecuencia, permite aislar las partes del programa que permanecen inalterables en el tiempo. La integridad que dan los objetos a los datos evita ambigüedades en su uso, dando mayor seguridad en los resultados.

Reusabilidad: Los programas que posean los mismas estructuras de información reutilizan las definiciones de objetos empleadas en otros programas, e incluso los procedimientos que los manipulan. Un desarrollo realizado con el modelo orientado a objetos es más fácil de mantener y de reutilizar.

Las características de la programación orientada a objetos que posibilitan estas ventajas sobre otros paradigmas de programación son:

La **ENCAPSULACIÓN** es la cualidad de unificar los datos y la forma de manipularlos, de esta forma podemos ocultar el funcionamiento de una clase y exponer solo los datos que manipula (mediante propiedades), así como proveer de medios para poder manipular dichos datos (mediante métodos). De esta forma solo exponemos al mundo exterior la información y la forma de manipularla, ocultando los detalles usados para manejar esos datos y, lo que es más importante, evitando que nadie manipule de una forma no controlada dicha información.

La **HERENCIA** es la cualidad de poder crear nuevas clases (o tipos) basadas en otras clases, de forma que la nueva clase obtenga todas las características de la clase que ha heredado, tanto los datos que contiene como la forma de manipularlos, pudiendo añadir nuevas características e incluso cambiar el comportamiento de algunas de las incluidas en la clase base, (siempre que así se haya previsto). Mediante la herencia podemos crear de forma fácil una jerarquía de clases que comparten un mismo comportamiento básico pero que cada nueva generación puede tener (y de hecho tiene) un nuevo comportamiento.

El **POLIMORFISMO** es la cualidad de implementar de forma particular algunas de las características que tienen las clases, de forma que cuando necesitemos usarlas no nos preocupe la implementación interna que cada una tenga, lo que realmente nos interesa o nos debe importar es que podemos usar esas características e incluso podamos acceder a ellas de forma anónima.

La **ABSTRACCIÓN** denota las características principles de un objeto que lo distinguen de todos los demás tipos de objetos y proporciona así fronteras conceptualmente nítidas.[Navarro07].

3.2. Fases del Proyecto

El proceso que hemos seguido para la consecución del proyecto consta de las siguientes fases:

a. Fase I: Estudio de las Estructura de la Aplicación.
b. Fase II: Estudio de las Funciones del Código de Aplicación.
c. Fase III: Estudio del Código de Aplicación.
d. Fase IV: Diseño de clases e interfaces.
e. Fase V: Modularización del programa.

Las tres primeras fases, se identifican con la fase definida por Sommerville como Fase de Ingeniería Inversa.

Durante cada una de estas tres fases, analizamos el programa creado por Earl Cox y extraemos información, la cual nos ayuda a documentar la organización y funcionalidad.

Es el proceso de analizar el software con el objetivo de recuperar su diseño y especificación.

La decisión de establecer estas fases, viene tomada porque el modelo del sistema borroso creado por el autor, consta de dos componentes principales: El código de aplicación que contiene el razonamiento lógico actual y un grupo de bloques de control y estructuras que gestionan el entorno de modelado actual. Además se incluyen un conjunto de funciones que implementan la funcionalidad necesaria para el funcionamiento del sistema.

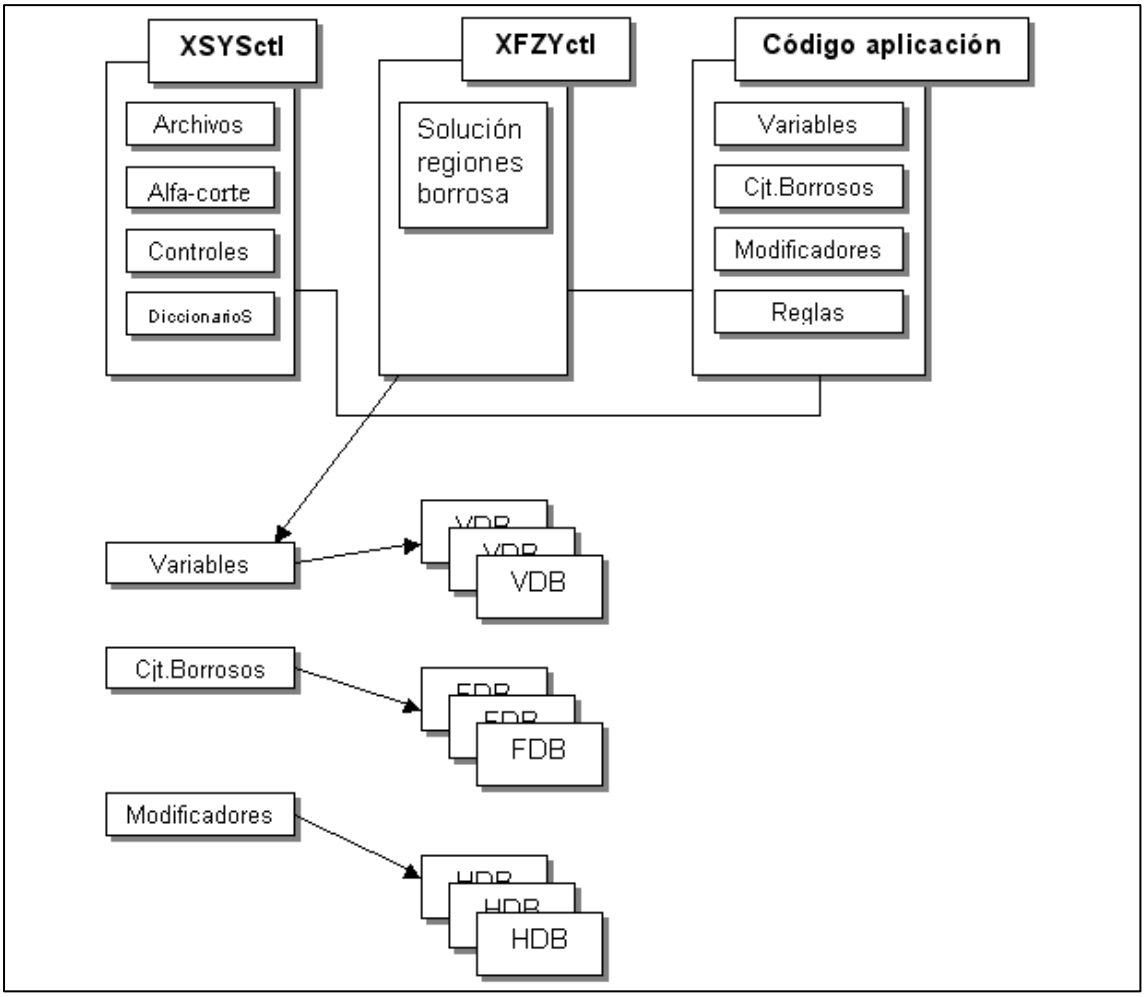
La fase de diseño de clases e interfaces, la identificamos con la Fase de Mejora de la Estructura del Programa, de Sommerville, en esta fase se diseña la aplicación, una vez se han identificado y estudiado las estructuras de control y funciones del programa del programa heredado.

La fase quinta fase la identificamos con la fase del mismo nombre de Sommerville, en esta fase reorganizamos el programa de forma que partes relacionadas se integren de forma conjunta. Lo que facilita la eliminación de componentes redundantes y mejora la comprensión **.**

3.2.1. Fase I : Estudio de las Estructuras de la Aplicación

Bloques de Control y Estructuras

Las estructuras de control de la aplicación de Cox se denominan XSYSCtl y XFZYctl. La primera es usada a lo largo de entorno de modelado y debe ser conectado al modelo de la aplicación en todos los casos. Esta estructura guarda la información sobre la localización del modelo, de los archivos de errores y de trazas, de los valores de las variables globales como el alfa-corte y para los modelos más complejos, una serie de tablas hash [1] que contienen el diccionario de varios componentes del modelo. La estructura de control XZFYctl tiene un papel principal para aproximar el conocimiento. Almacena la información sobre cada una de las variables solución del modelo. Cuando una variable (representada por un bloque de definición de variable [VDB]) es añadido a esta estructura, el modelo automáticamente destina una área de trabajo (representada por un conjunto borroso) e inicializa los métodos asociados para la correlación, la implicación y la defuzificación.



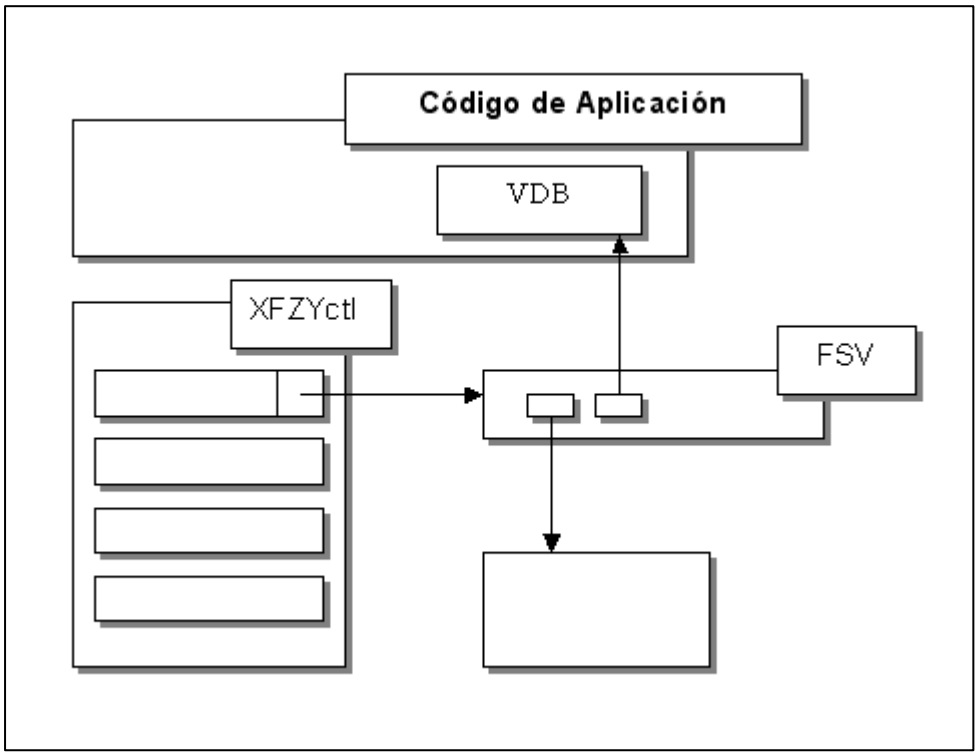
El código de la aplicación se compone de variables borrosas, de conjuntos borrosos, de modificadores y de reglas. A continuación se explican estos componentes:

Variables borrosas

Cada regla en un modelo borroso especifica la relación entre una región borrosa (antecedente) y una región borrosa (consecuente) asociado con una de las soluciones variables. Cada vez que una regla es evaluada, el espacio

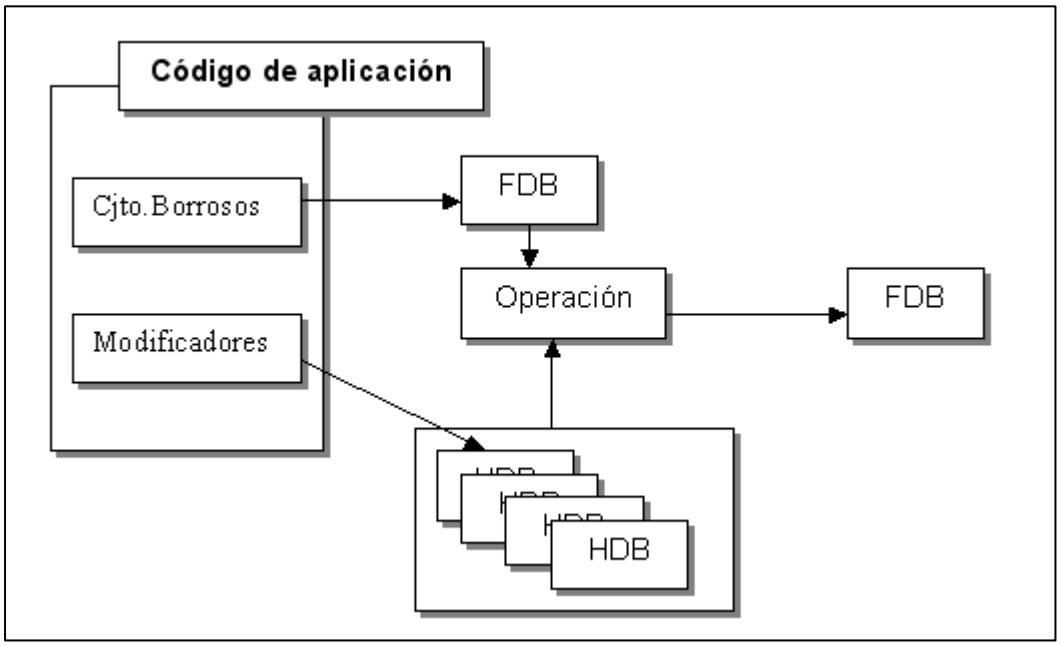
asociado a la variable solución es modificado (siempre y cuando la verdad sea distinta de cero).

El bloque de control XFZctl tiene espacio para 32 variables para un modelo. Cada espacio tiene un puntero a una estructura donde se almacena la solución (FSV). Esta estructura contiene punteros al bloque de definición de la variable (VDB) y al bloque de descripción del conjunto borroso (FDB). Cuando un modelo borroso es inicializado un VDB debe ser destinado para cada variable solución e insertado al bloque XFZYctl.



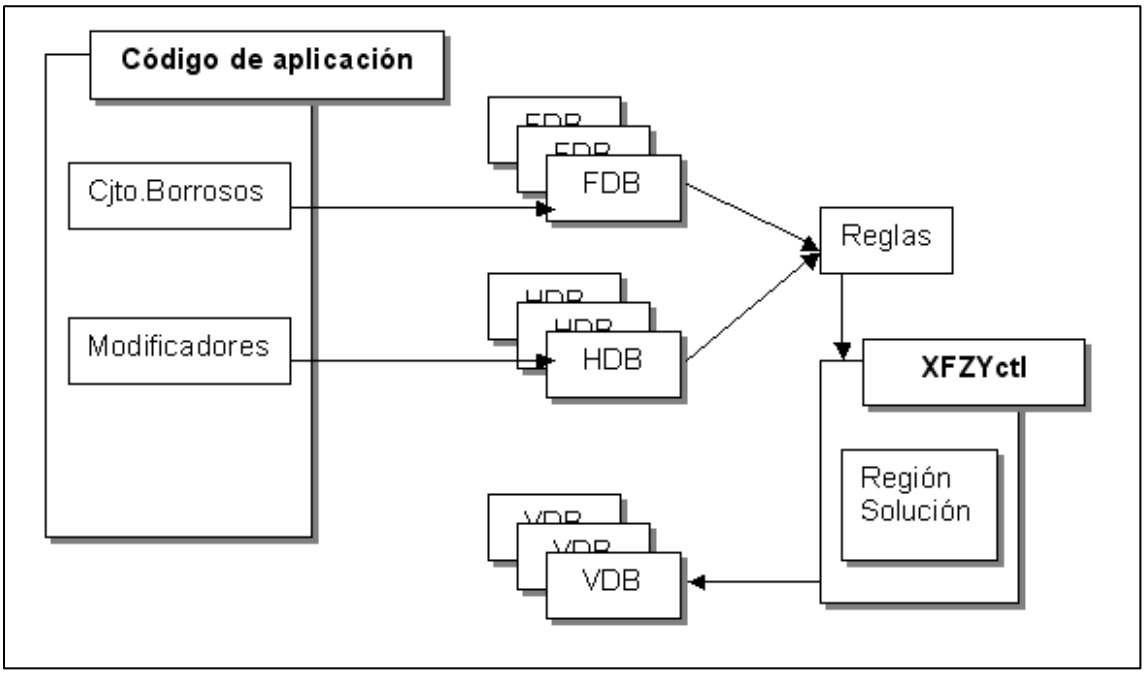
Modificadores

Un modificador "cambia" la forma de un conjunto borroso. Cada modificador es descrito por un bloque descriptor de modificadores (HDB) y su función depende del tipo del mismo. El resultado de aplicar un modificador a un conjunto borroso es otro conjunto borroso.



Reglas

Una regla en el modelo borroso no es almacenada como una estructura, aplica cualquier modificador, determina la pertenencia de un escalar a un conjunto borroso, realiza AND o OR operaciones y usa la verdad del predicado según sea una proposición condicional o una proposición no condicional.



Las reglas incondicionales simplemente combinan el conjunto borroso consecuente con la actual región borrosa. En el caso de las reglas condicionales el proceso es más complejo. Una regla condicional necesita la verdad del antecedente para poder aplicar el método de correlación correcto. Cada modelo mantiene en un vector de texto las reglas lógicas en la forma estándar IF-THEN.

Alfa-cortes

El umbral del alfa-corte controla el lanzamiento de las reglas así como la salida de las operaciones de los conjuntos borrosos. Un alfa-corte hace que todos los valores de la función de pertenencia a partir de un valor sean igual a cero. Cuando un alfa-corte es aplicado a la verdad de una regla, ésta determina si el valor de pertenencia es lo suficientemente grande para lanzar la regla. Cuando un alfa-corte es aplicado a un conjunto borroso, se establece una línea a través de la función de pertenencia igual al alfa-corte. Los valores de pertenencia por debajo de esta línea son considerados igual a cero.

Existen varios umbrales de alfa-corte en el modelo borroso analizado. Cada uno de los niveles de alfa-cortes controla alguna función del protocolo de modelado.

El umbral almacenado en el bloque XSYSctl (en el atributo XSYSalfacut) es el valor primario del umbral y está inicializado a cero cuando la función MdlConnettoFMS es lanzada. Este valor es utilizado por las funciones FzyCondProposition y FzyUnCondProposition para decidir si un predicado es cierto.Cuando los conjuntos borrosos (FDB's) y las variables (VDB's) son inicializados, su umbral de alfa-corte es heredado del valor del atributo XSYSalfacut en XSYSctl.

El alfa-corte almacenado en el FDB es usado para controlar el vector de valores de pertenencias que resultan de aplicar las operaciones de intersección, unión y negación de conjuntos.

El alfa-corte almacenado en el FSV es heredado del VDB cuando es añadido al área de trabajo (usando FzyAddFZYctl). Este valor es asignado al conjunto borroso de salida cuando se llama al proceso de defuzzificación.

3.2.2. Fase II : Estudio de las Funciones del Código de la Aplicación

Funciones sobre Conjuntos Borrosos

FzyGetMembership: Esta función calcula el grado de pertenencia de un valor dado como parámetro al conjunto borroso.

FzyEquivalentScalar: Esta función busca en el array de pertenencia del conjunto borroso especificado como parámetro hasta que encuentra una pertenencia que sea igual o menor que un grado de pertenencia especificado como parámetro.

FzyGetScalar: Un grado de pertenencia a un conjunto borroso puede estar asociado con uno o más valores del domino. La función encuentra el escalar asociado a un valor de pertenencia, más cercano al límite inferior del dominio del conjunto borroso.

Todos los conjuntos borrosos en un modelo deben estar en forma normalizada, sino complicaría la tarea de verificar el modelo y aplicarle metricas de estabilidad.

FzyIsNormal: Función que indica si un conjunto borroso está o no en forma normal.

FzyGetHeight: Función que devuelve el mayor valor de pertenencia de un conjunto borroso, lo cual implica un importante papel cuando tratamos de medir cuan bueno es un modelo y su compatibilidad con los datos.

FzyNormalizeSet: Función que normaliza un conjunto borroso encontrando la máxima altura en él y usando esto para escalar el array de grados de pertenencia.

Funciones sobre Curvas

Estas funciones generan el vector de pertenencia de los elementos del dominio de un conjunto borroso, vector donde se almacena el grado de pertenencia de cada elemento, que variará en función del tipo de curva que represente la función de pertenencia del conjunto.

FzyLinearCurve
FzySCurve
FzyPiCurve
FzyBetaCurve
FzyGaussianCurve
FzyTriangleCurve
FzyShoulderedCurve

FzyFuzzyScalar: Esta funcion convierte un rango, definido por dos escalares o una variable escalar en un número borroso. Se puede seleccionar una representación triangular, pi-curva, beta-curva…La anchura y los puntos de

inflexión son calculados automaticamente de la magnitud del escalar o del dominio dado.

Funciones sobre α-cortes

FzyAboveAlfa: Esta función comprueba si un elemento del dominio del conjunto borroso, tiene un valor mayor que el valor del corte- alfa.

FzyApplyAlfa: Esta función aplica el corte-alfa a el dominio del conjunto borroso, generando un nuevo dominio restringido a los elemento que están por encima del alfa-corte, los que no lo están el valor de pertenencia al conjunto borroso se actualiza a cero.

Funciones sobre Operadores

FzyAND: Esta función implementa el operarador AND básico de Zadeh, tomando el mínimo de dos valores de pertenencia correspondientes a dos conjuntos borroso.

FzyOR: Esta función implementa el operarador OR básico de Zadeh, tomando el máximo de dos valores de pertenencia correspondientes a dos conjuntos borroso.

FzyApplyAND: Función que implementa la intersección de dos conjuntos borrosos que es a menudo usada como una nueva región borrosa en un modelo.

FzyApplyOR: Esta función implementa la unión de dos conjuntos borrosos que es a menudo usada como una nueva región borrosa en un modelo.

FzyCompAND, FzyCompOR: La gran variedad de operadores compensatorios para la unión o intersección de conjuntos borrosos, están recogidos en estas dos funciones. La funciones soportan los tipos de operadores algebraicos básicos así como la compensación de Yager.

FzyApplyNOT: Esta función reune en una, los diferentes operaciones de complemento o negación de un conjunto borroso.

Funciones sobre Modificadores

FzyApplyHedge: Esta función administra la aplicación de modificadores, diferenciando el código en dos partes, una primera sección que soporta modificadores definidos por el usuario y otra que administra los modificadores existentes en el modelo.

Funciones sobre Métodos de Inferencia

FzyMonotonicLogic: Esta función calcula el grado de pertenencia de un valor pasado como parámetro a un conjunto borroso y después, utiliza la pertenencia para seleccionar un valor del dominio de un conjunto borroso solución.

FzyCondProposition: Esta función encapsula varios métodos de implicación, no siendo aquí donde se evalua la propsicion, sino en una función que la invoque.
Se selecciona una técnica de correlación, se genera un conjunto borroso solución para el consecuente y se actualiza la variable solución en le bloque de control.

Funciones sobre Métodos de Correlación

FzyCorrMinimun, FzyCorrProduct: Estas funciones reducen la altura de un conjunto borroso especificado por parámetro truncando o escalando la

grado de pertenencia; empleando para ello el mecanismo de implicación mini-max, trucando la variable consecuente con el valor de verdad el antecedente.

Funciones sobre Métodos de Defuzzyficación

FzyDefuzzify: Esta función encapsula todas las técnicas de defuzzyficación en una única función. La función devuelve un valor de tipo double como valor esperado del conjunto borroso solución y el método de defuzzyficación a aplicar es un valor pasado como parámetro.

3.2.3. Fase III : Estudio del Código de la Aplicación

En el programa principal (main) todas la estructuras de políticas del modelo están localizados y almacenadas en el diccionario de la estructura XSYSctl.

Definición de las variables solución

Una variable solución aparece en el lado izquierdo del consecuente de una regla y es importante a la hora de realizar el proceso de defuzificación después de que el modelo de política se ha ejecutado. En las proposiciones condicionales de la forma, if x es Y then z es W, la z es la solución. Después de que todas las reglas en las que el consecuente use esa variable hayan sido evaluadas, la defuzificación es usada para encontrar el valor esperado de la variable solución. Cada política puedecontener un máximo de 32 variables solución. Cada una de las variable solución está representado por un bloque descriptor de variable (VDB) . Esta estructura es insertada en el area de trabajo de la estructura XFZYctly es gestionada por las proposiciones tanto condiciones como por las incondicionales.

Crear y almacenar conjuntos borrosos en el código de aplicación

Los conjuntos borrosos pueden ser almacenados de dos formas distintas:en el código del programa o en el diccionario de las políticas(PDB). La forma más fácil de usar conjuntos borrosos, al menos para modelos de poca complejidad, es através de vectores de FDB (bloque descriptor de conjuntos borrosos).

Crear y almacenar conjuntos borrosos en el diccionario de una política

Las políticas proveen de un mecanismo de almacenamiento, mecanismos para compartir y componentes de modelado para ser usados en modelos comlejos. El trabajo con políticas disminuye el código de aplicación ya que este código está dentro de la estructura de la política.

Cargar y crear modificadores

Los modificadores como los conjuntos borrosos pueden ser almacenados en el código de la aplicación (en vectores de estructuras de HDB) o en las estructuras de PDB formando parte del diccionario de la política.

Para cargar los modificadores predefinidos en nuestro modelo, usamos dos funciones. La función MdlInsertHedges instala los modificadores en el diccionario de una política específica. La función FzyInsertHedges carga los modificadores en un vector de estructuras HDB que hayamos especificado.

Para encontrar un modificador concreto se debe buscar a lo largo del vector hasta encontrar el modificador que concuerda con el identificador

Uno de los problemas que conlleva esta búsqueda es la sobrecarga que produce que irá en aumento según aumentemos el número de modificadores incluidos como predefinidos.

Se tiene la posibilidad de que el usuario defina sus propios modificadores que serán reconocidos por la función FzyApplyHedge. Cuando se crea un nuevo modificador éste uede ser almacenado en una política (dentro del diccionario de la misma) o en una variable en el código de la aplicación

Establecer la política del entorno

Existen dos funciones para establecer el entorno de la política. Si el programa principal ha reasignado un valor al campo de XSYScurrPDBtr (atributo de la estructura XSYSctl) con la dirección de la política a utilizar antes de la llamada a la función, sólo se necesita para establecer el entorno de la función prcPDBpt r= XSYSctl.XSYScurrPDBptr, donde prcPDBptr es un puntero que contiene la dirección de la política a usar.

Existen casos en los que no es posible conocer la política y sólo tenemos el valor del identificador. En estos casos se hace necesario la utilización de la función MdlFindPDB que busca la política dentro de un vector donde están almacenadas las mismas mediante el identificador.

Inicializar el área de trabajo de lógica borrosa para la política

Cada política comparte el bloque de control XFZYctl. En esta estructura se encuentran los conjuntos con los cuales estamos trabajando y otros parámetros para cada una de las variables en la política. Si bien es verdad que este bloque es utilizado por varios módulos, cada vez que un módulo inicialice una política deberá inicializar el bloque de control XFZYctl.

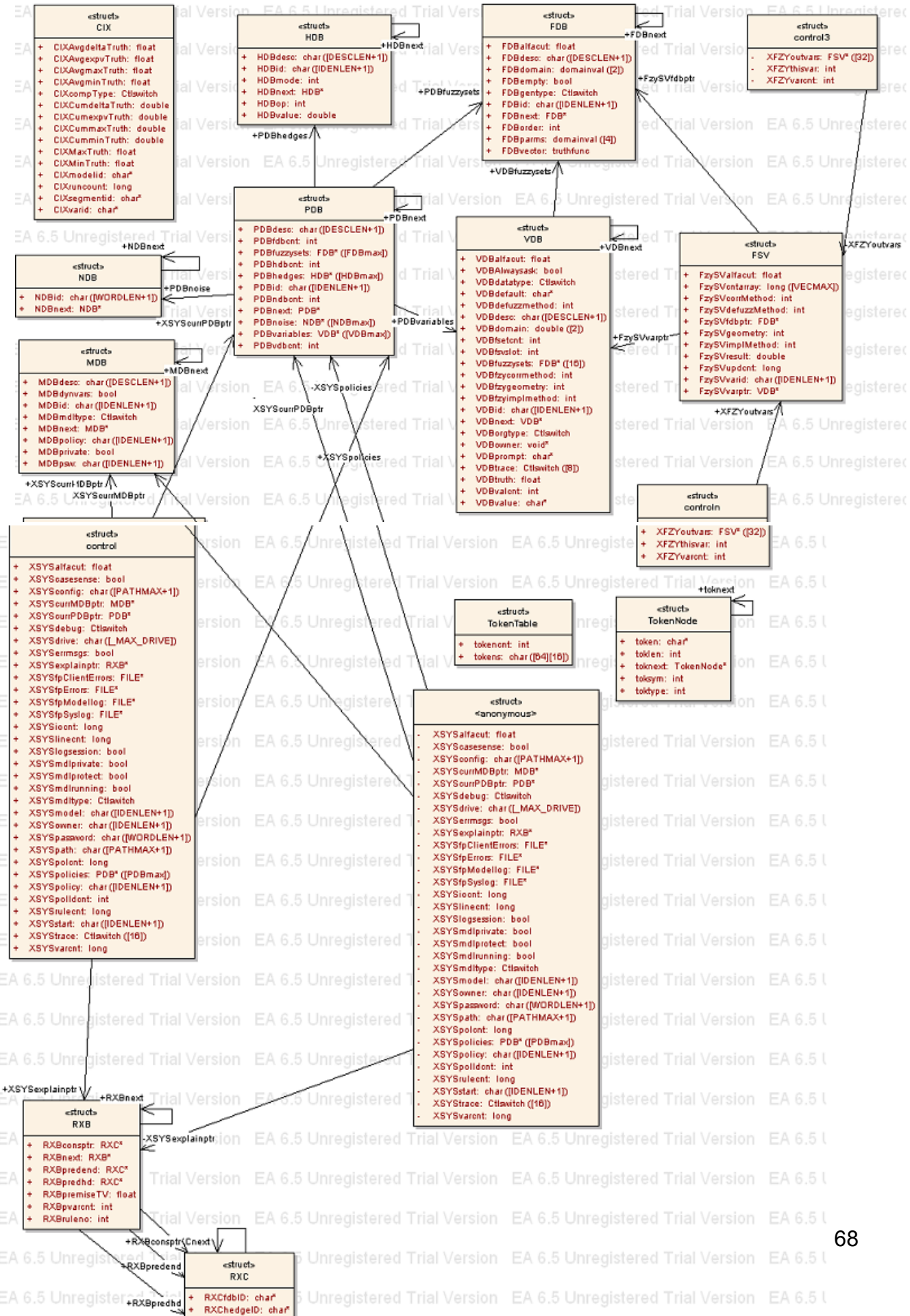
Localizar los conjuntos borrosos y modificadores necesarios

Una de las estructuras asociada a las políticas son las que permiten localizar y almacenar. Se puede buscar y almacenar las direcciones de cada

conjunto borross y modificador usado en la política como ya se ha visto en los anteriores puntos.

Si no se usan este mecanismo de las políticas existen diversas opciones a elegir: definir los conjuntos borrosos y los modificadores dentro del código de la política, almacenar los conjuntos borrosos en un vector estático externo, o pasar los conjuntos borrosos y los modificadores en a la política a través de un vector de parámetros.

3.2.4. Diseño obtenido de las fases anteriores



3.2.5. Fase IV: Diseño de las clases e interfaces

INTERFACES DEFINIDAS EN EL MODELO

Cuando uno de nuestras tareas es conseguir el polimorfismo, ineludiblemente tenemos que hablar de las interfaces, ya que, principalmente, nos posibilita utilizar esta característica de la POO. Las interfaces son una clase especial en la que solamente se definen los métodos y propiedades que una clase que la implemente debe codificar. Las interfaces representan un contrato, de forma que cualquier clase que la implemente debe utilizar los miembros de la interfaz usando la misma forma en que ésta la ha descrito: mismo número de argumentos, mismo tipo de datos devuelto, etc.

Gracias a la implementación de interfaces podemos crear relaciones entre clases que no estén derivadas de la misma clase base, pero que tengan métodos comunes, al menos en la forma, aunque no necesariamente en el fondo.

Además las interfaces, hacen que los subsistemas de diseño sean más fáciles de utilizar, evitando así el acoplamiento entre estos, mientras que cada subsistema implementa sus responsabilidades. [Navarro07].

Interfaz Hedge (Modificadores)

Define los métodos que debe implementar cualquier clase que implemente la interfaz Hedge.

En función del comportamiento de un objeto modificador o hedge con el resto de los objetos del modelo se definen dos métodos:

Método getNombre()

Devuelve una cadena de caracteres que representa el nombre del hedge o modificador. Los hedges en el modelo son construcciones lingüísticas o palabras que representan conceptualmente el grado en el que modifican un conjunto borroso.

Método aplicarHedge(double valor)

Los hedges o modificadores transforman un conjunto borroso en otro nuevo. El grado en que la superficie borrosa de un conjunto –representado por su curva– es transformada y la naturaleza de esa transformación no están basados en una teoría matemática de la topología de la superficie borrosa, sino en el ajuste de la percepción de lo que supone la transformación.

El método recibe un valor que pertenece al dominio del conjunto borroso y devuelve el valor de su transformación.

La definición original del hedge *"very"* según Lotfi Zadeh intensifica la superficie del conjunto borroso elevando al cuadrado el grado de pertenencia de cada punto del conjunto a la función - $\mu^2[x]$ - y la definición del hedge *"somewhat"* diluye el espacio borroso tomando la raíz cuadrada del grado de pertenencia de cada punto de la función - $\mu^{1/2}[x]$ -.

Así la clase que defina el hedge very en su implementación del método aplicarHedge calculará el cuadrado de un elemento perteneciente al conjunto borroso, mientras que en la clase que defina el hedge somewhat calculará la raíz cuadrada.

Se seguirá el mismo procedimiento para definir nuevos hedges y todos deben implementar los mismos métodos atendiendo a sus particularidades.

La mejora sobre el código heredado supone no tener que modificar el código de la aplicación cada vez que se define un nuevo hedge o se modifica la implementación de uno ya existente.

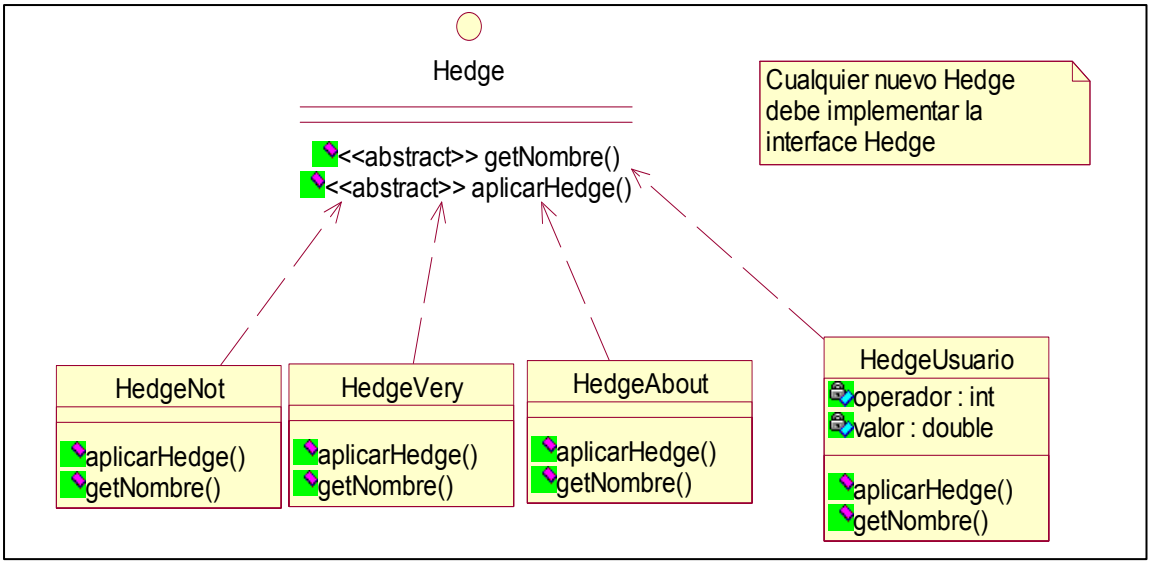


Diagrama de clases que muestra el diseño propuesto para los Modificadores

Interfaz Operador

Define los métodos que debe implementar cualquier clase que implemente la interfaz Operador, y cualquier nuevo operador que haya de ser incluido en el sistema.

Método Abstracto getNombre()

Devuelve una cadena de caracteres que representa el nombre del operador.

Método Abstracto aplicarOperador(double op1, double op2)

Devuelve el valor de aplicar a dos operandos, que vienen dados como parámetros, el operador específico de cada clase que implementa la interfaz.

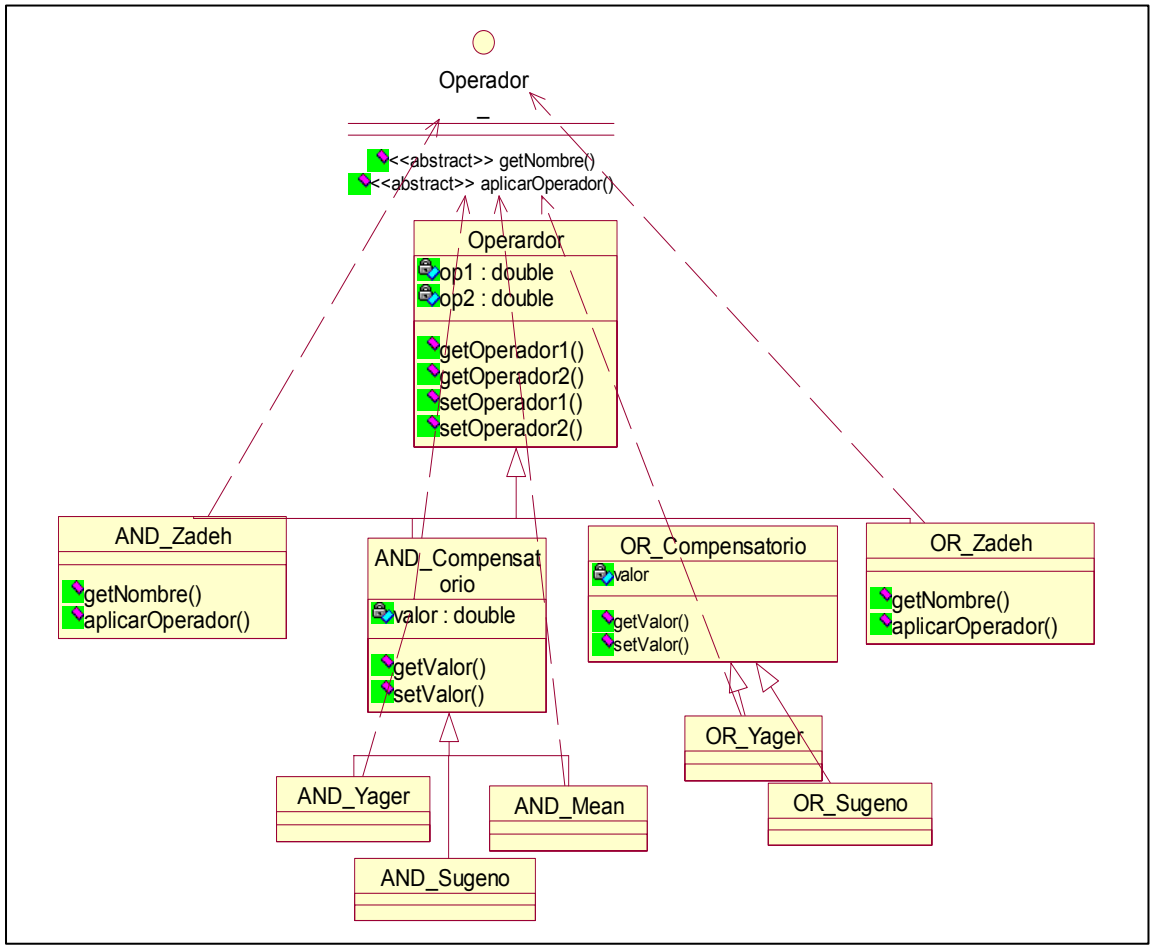


Diagrama de clases que muestra el diseño propuesto para los Operadores.

Interfaz Defuzzyficador

Define los métodos que debe implementar cualquier clase que implemente la interfaz Defuzzyficador.

Método getNombre()

Devuelve una cadena de caracteres que representa el nombre del defuzzyficador.

aplicarDefuzzyficador(ConjuntoBorroso cb)

Devuelve el valor resultante de aplicar un método de defuzzyficación a un conjunto borroso.

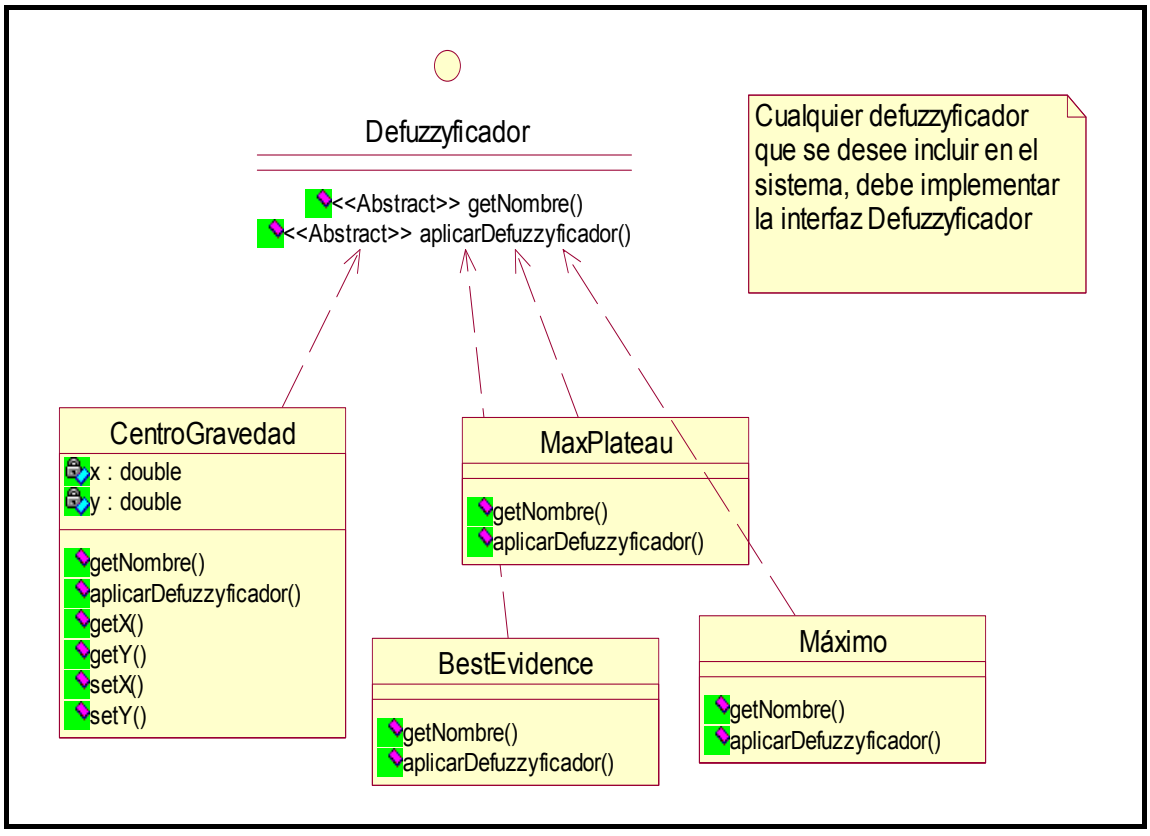


Diagrama de clases que muestra el diseño propuesto para los Defuzzyficadores

CLASES

En la programación orientada a objetos los programas se organizan como colecciones cooperativas de objetos, cada una de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembro de una

jerarquía de clases, unidas fundamentalmente mediante relaciones de herencia. [Navarro07].

Las clases son plantillas donde se definen un conjunto de campos y métodos que podrán ser ejecutados por los objetos de esa clase.

Clase Conjunto Borroso

Atributos de la clase ConjuntoBorroso:

Nombre: Nombre que identifica el conjunto borroso.
Mínimo: Valor del límite inferior de los elementos del conjunto borroso.
Máximo: Valor del límite superior de los elementos del conjunto borroso.
Curva: Representación gráfica del conjunto borroso y determinante del valor del grado de pertenencia de los elementos al conjunto borroso.

Métodos Accesores y Modificadores:

getNombre()

Devuelve el nombre del conjunto borroso.

getMinimo()

Devuelve el valor del límite inferior del conjunto borroso.

getMáximo()

Devuelve el valor del límite superior del conjunto borroso.

getCurva()

Devuelve el atributo curva del conjunto borroso.

setNombre(string nombre)

Modifica el valor del atributo nombre con uno del mismo tipo pasado por parámetro.

setMínimo(double min)

Modifica el valor del atributo mínimo con uno del mismo tipo pasado por parámetro.

setMáximo(double max)

Modifica el valor del atributo máximo con uno del mismo tipo pasado por parámetro.

setCurva(Curva curva)

Modifica el valor del atributo curva con uno del mismo tipo pasado por parámetro.

Otros métodos:

Método obtenerPertenencia(double valor)

Devuelve el valor del grado de pertenencia al conjunto borroso de un valor dado por parámetro.

Método obtenerEscalar(double grado)

Devuelve el valor del escalar asociado a un valor de grado de pertenencia.

Método copiar()

Devuelve un objeto de tipo conjunto borroso copia del actual.

Clase Abstracta Curva

Una clase abstracta tienen al menos un método abstracto. Un método abstracto es aquel sin implementación real. El objetivo de tener una clase abstracta es permitir la declaración de métodos, aunque estos estén sin implementar y difiere de las interfaces en que puede contener atributos y métodos no abstractos; son las subclases de estas clases abstractas las encargadas de implementar todos los métodos abstractos.

Las curvas o funciones de pertenencia son, como se ha comentado en capítulos anteriores, la representación gráfica de un conjunto borroso y determinan el grado de pertenencia de un elemento dado a dicho conjunto.

Existen tipos diferentes de curvas pero todas tienen parámetros en común que es necesario definir, en base a esto se crea una clase abstracta curva de la que derivan las demás clases curva y que implementarán para cada caso concreto el método aplicarCurva y que además dependiendo del tipo a tratar necesitarán o no parámetros adicionales.

Entre los tipos de curvas encontramos:

- Curva Lineal
- Curva Sigmoidea
- Curva Pi
- Curva Beta
- Curva Gaussiana

Atributos de la clase Curva:

Minimo: Límite inferior de la curva, dado por el límite inferior del conjunto borroso al que pertenece.

Máximo: Límite superior de la curva, dado por el límite superior del conjunto borroso al que pertenece.

Métodos Accesores y Modificadores:

getMinimo()

Devuelve el límite inferior de la curva.

getMáximo()

Devuelve el límite superior de la curva.

setMínimo(double min)

Modifica el valor del atributo mínimo con uno del mismo tipo pasado por parámetro.

setMáximo(double max)

Modifica el valor del atributo máximo con uno del mismo tipo pasado por parámetro.

Métodos Abstractos:

funcionPertenencia(double valor)

Devuelve el valor del grado de pertenencia de un valor pasado por parámetro, al conjunto borroso. Cada clase que herede de la clase abstracta curva ha de implementarlo.

Clase CurvaLineal

Hereda de la clase abstracta curva sus atributos y métodos.

Además contiene otro atributo de nombre tipo que define el tipo de curva lineal, que como hemos visto secciones anteriores puede ser INCREASE ó DECREASE.

Métodos Accesores y Modificadores:

getTipo()

Devuelve el tipo de la curva lineal.

setTipo()

Modifica el atributo tipo con otro valor dado por parámetro del mismo tipo

funcionPertenencia(double valor)

Devuelve el valor del grado de pertenencia de un valor pasado por parámetro, al conjunto borroso.

Clase CurvaS

Hereda de la clase abstracta curva sus atributos y métodos.

Además contiene un atributo de nombre Tipo, que define el tipo de curva S, que como hemos visto secciones anteriores puede ser GROWTH ó DECLINE; y un atributo Inflexion, que determina el valor del punto de inflexión necesario para los cálculos en las curvas S.

Métodos Accesores y Modificadores:

getTipo()

Devuelve el tipo de la curva S.

getInflexion()

Delvuelve el valor del punto de inflexión de la curva.

setTipo(int tipo)

Modifica el atributo tipo con otro valor dado por parámetro del mismo tipo.

setInflexion(double inflexion)

Modifica el atributo inflexion con otro valor dado por parámetro y del mismo tipo.

funcionPertenencia(double valor)
Devuelve el valor del grado de pertenencia de un valor pasado por parámetro, al conjunto borroso.

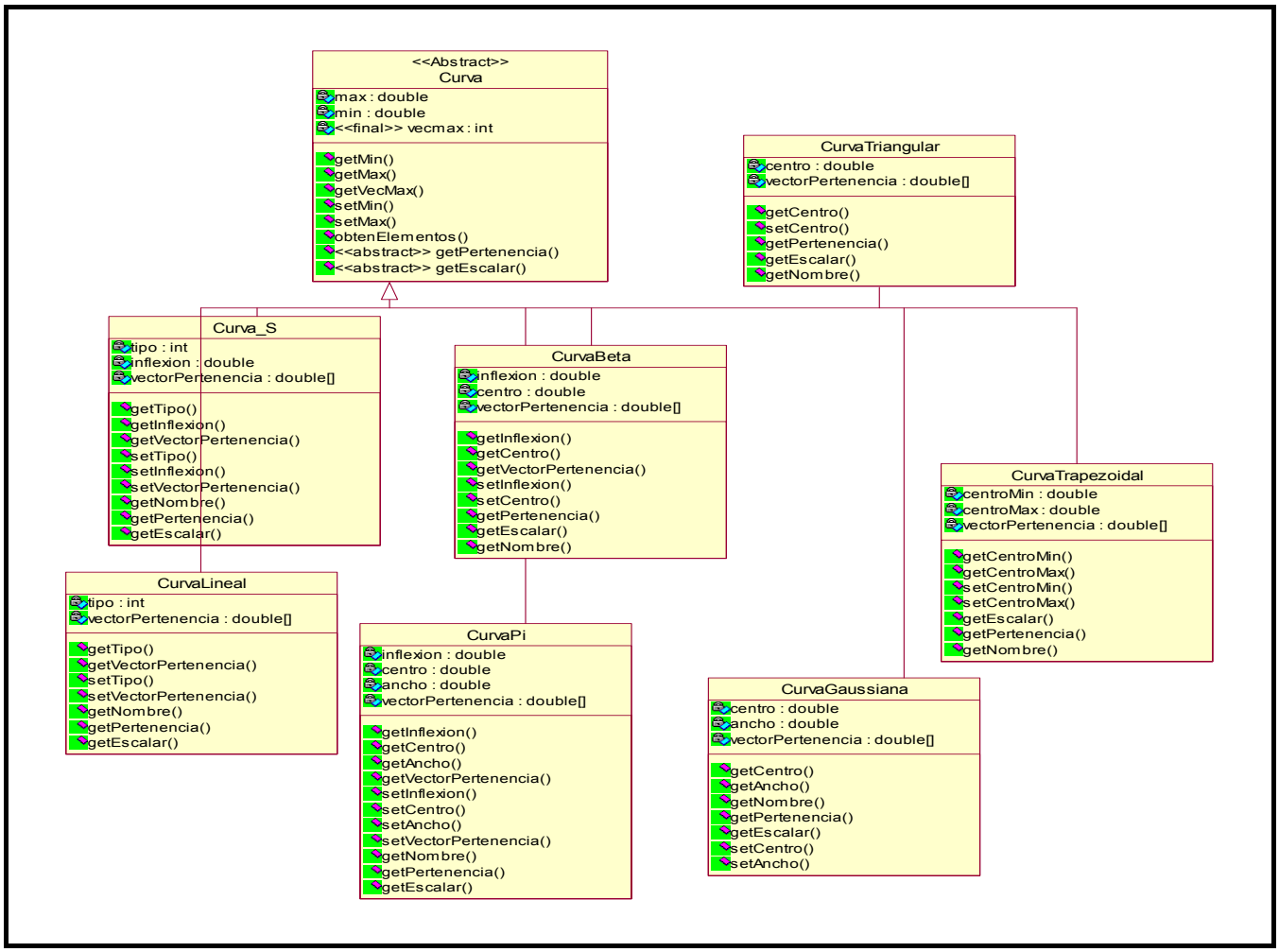


Diagrama de clases que muestra el diseño propuesto para las Curvas.

Clase Variable Lingüística

En el contexto de la aplicación creada por Earl Cox, el concepto de variable lingüística no difiere del de conjunto borroso, se refiere indistintamente con uno u otro nombre.

Para una mayor facilidad de comprensión en nuestro modelo, las variables lingüísticas estarán formadas por uno o más conjuntos borrosos. Ilustramos este hecho con un ejemplo sencillo, la variable lingüística Temperatura, que identifica el fenómeno que estamos modelizando, está formado por los conjuntos borrosos Caliente, Frío y Templado.

Atributos de la clase Variable Lingüística:

Nombre: Nombre de la variable lingüística, que representa el fenómeno a modelizar.
Mínimo: Límite inferior de los valores que contemplamos en la definición de la variable.
Máximo: Límite superior de los valores que contemplamos en la definición de la variable.
Tabla: Estructura que almacena los conjuntos borrosos que definen la variable lingüística.

Métodos Accesores y Modificadores:

getNombre()

Devuelve el nombre de la variable lingüística.

getMínimo()

Devuelve el límite inferior de la variable lingüística.

getMáximo()

Devuelve el límite superior de la variable lingüística.

getTabla()

Devuelve la estructura que contiene los conjuntos borrosos de la variable.

setNombre(string nombre)

Modifica el valor del atributo nombre con uno del mismo tipo pasado por parámetro.

setMinimo(double minimo)

Modifica el valor del atributo mínimo con uno del mismo tipo pasado por parámetro.

setMáximo(double máximo)

Modifica el valor del atributo máximo con uno del mismo tipo pasado por parámetro.

setTabla(Tabla tabla)

Modifica el valor del atributo tabla con uno del mismo tipo pasado por parámetro.

Clase Alfa-Corte

La clase alfaCorte es la plantilla para los objetos del mismo nombre.

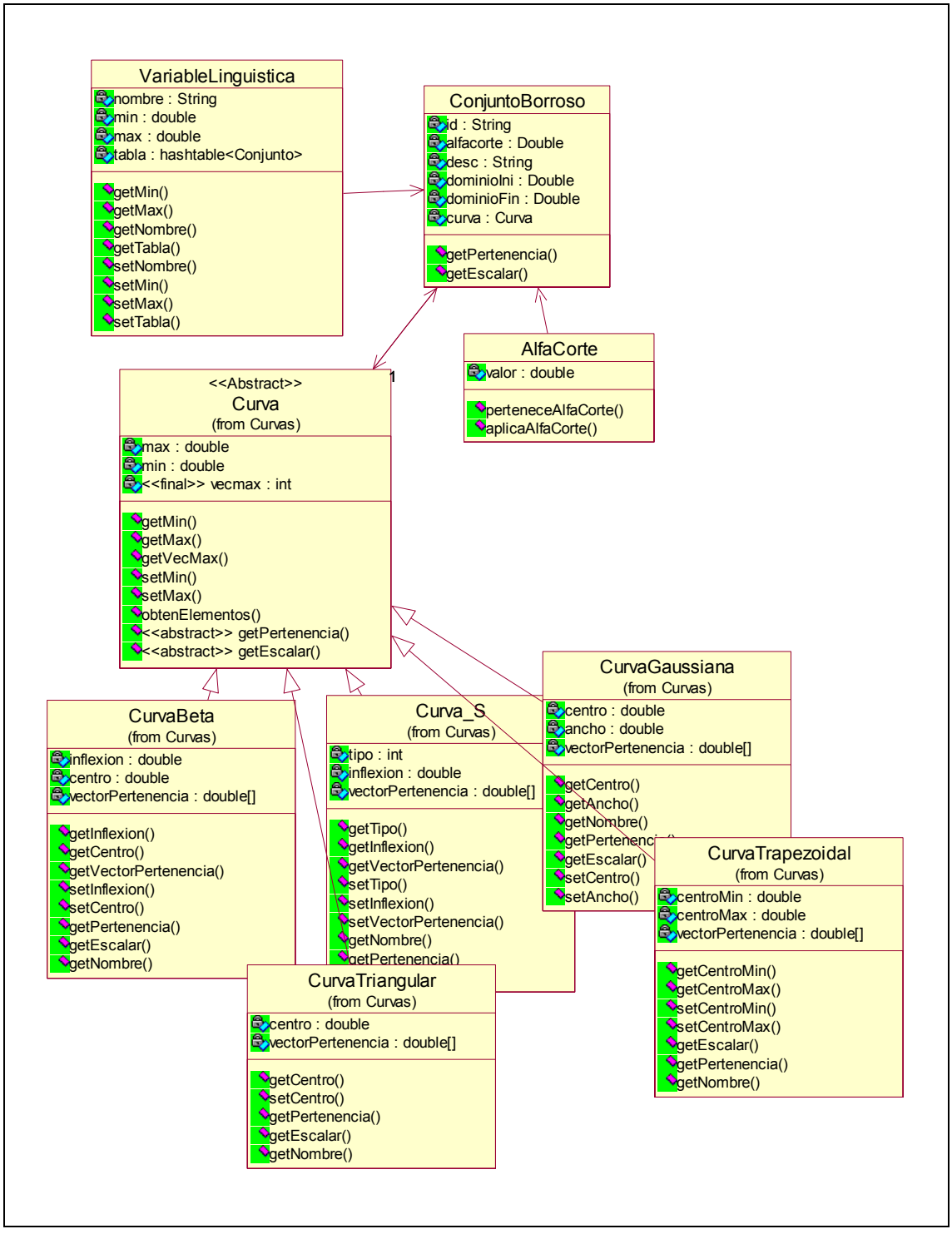
Método generaAlfaCorte()

Este método aplica el corte-alfa a el dominio del conjunto borroso, generando un nuevo dominio restringido a los elemento que están por encima del alfa-corte, los que no lo están el valor de pertenencia al conjunto borroso se actualiza a cero.

Método perteneceAlfaCorte(double valor)

Devuelve cierto o falso, dependiendo de si el valor pasado como parámetro pertenece o no al conjunto borroso que determina el valor del alfa corte.

Este método es de gran utilidad para la evaluación del antecedente de las reglas de inferencia.



Clase Operador

La clase operador es la clase base para otras nuevas clases destinadas a aportar la funcionalidad de los operadores en el sistema. Haciendo que la tarea de añadir nuevos operadores o modificación de los ya existentes sea considerablemente más sencilla que en el modelo estructurado, donde esto mismo suponía modificación a nivel de código de los ficheros de los que se sirve la aplicación.

Método getOperador1(), getOperador2

Métodos accesores a los atributos de la clase.

Método setOperador1(), setOperador2

Métodos modificadores de los atributos de la clase.

Clase Expresion

La clase Expresion modeliza el concepto de lo que es una expresion en una regla de inferencia.

Como hemos visto en capítulos anteriores diferenciamos dos tipos de reglas, reglas condicionales e incondicionales, las reglas condicionales que es el caso más general están formadas por una expresión en el antecedente y otra en el consecuente.

La expresión del consecuente está formado por un parámetro y un conjunto borroso solución del modelo; sobre el parámetro queremos obtener un valor, obtenido de la aplicación del valor resultante de evaluar la expresión del antecedente ,en el conjunto borroso solución.

En base a estas consideraciones los atributos de la clase Expresion:

Parámetro: parámetro solución para el cual queremos obtener un valor solución.

Conjunto Solución: Conjunto sobre el cual se busca la solución una o más reglas de inferencia.

Métodos Accesores y Modificadores:

getParámetro()

Devuelve el valor del atributo parámetro.

setParámetro(double valor)

Modifica el valor del atributo parámetro.

getConjuntoSolucion()

Devuelve el valor del atributo conjunto solución.

setConjuntoSolucion(ConjuntoBorroso cb)

Modifica el valor del atributo conjunto borroso.

Clase ExpresiónAntecedente

La clase expresión antecedente modeliza las expresiones que aparecen en el antecedente de las reglas de inferencia de borrosa. Las cuales pueden estar formadas por una o más expresiones unidas por operadores.

Atributos de la Clase Expresión Antecedente:

listaExpresiones: lista de expresiones que aparecen en el antecedente, son del tipo Expresión.

listaOperadores: lista de operadores que aparecen en el antecedente y que aplican sobre los resultados de evaluar las expresiones.

Métodos de la clase Expresión Antecedente:

getListaExpresiones()

Devuelve el atribulo lista de expresiones.

getListaOperadorse()

Devuelve el atribulo lista de operadores.

setListaExpresiones(listaExpresiones listaEx)

Modifica el atribulo lista de expresiones.

setListaOperadorse(listaOperadores listaOp)

Modifica el atribulo lista de operadores.

Clase Base de Reglas

Modeliza el concepto de una base de reglas donde se almacenen las reglas de inferencia de un modelo y que a la vez pueda suministrar reglas al sistema.

Atributos de la clase Base de Reglas:

Tabla Antecedentes: Estructura que almacena para cada regla su antencedente, que será un objeto ExpresionAntencedente.

Tabla Consecuentes: Estructura que almacena para cada regla su consecuente, que será un objeto Expresion.

Métodos de la clase Base de Reglas:

Método dameRegla()

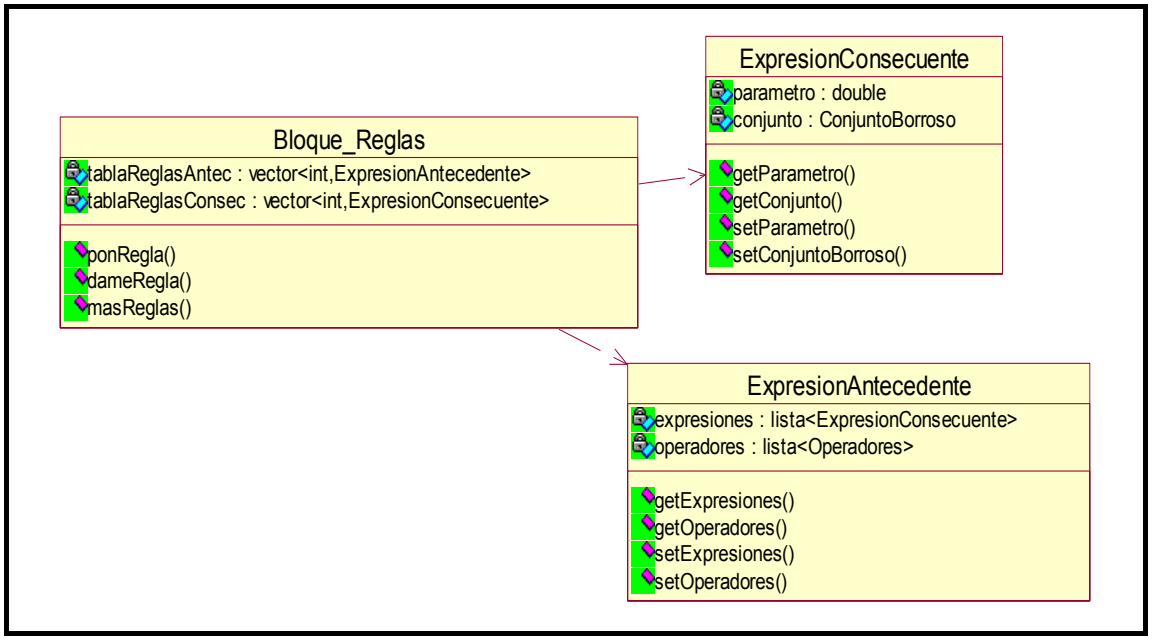
Proporciona al sistema una por unas las reglas almacenadas para un modelo.

Método ponRegla()

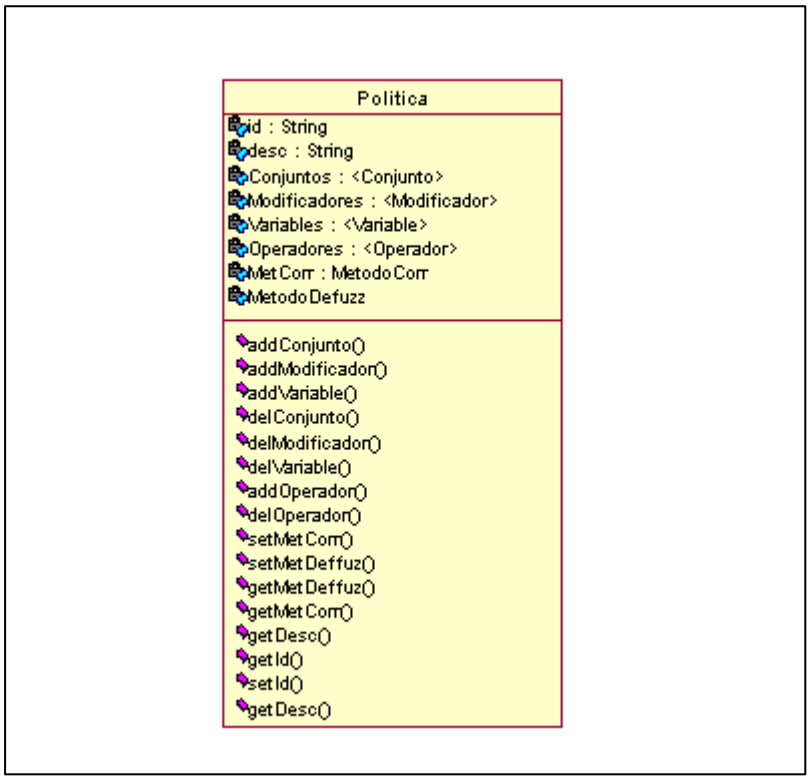
Almacena una regla en el orden correspondiente según el número de regla y con el formato requerido ExpresionAntecedente, ExpresionConsecuente si se trata de una regla condicional o Expresion Consecuente si se trata de una regla no condicional.

Método masReglas()

Devuelve cierto o falso si hay más reglas que han de ser proporcionadas al sistema según un modelo dado.



Clase Política



Esta clase contiene las implementaciones que representan a las políticas.

Las políticas representan el modo de comportamiento del sistema implementado, o lo que es lo mismo son las directrices que rigen la actuación del sistema. El sistema tendrá un comportamiento distinto dependiendo de los Operadores, Conjuntos, Variables, Modificadores y de los Métodos de correlación y de defuzzificación que incluya la política utilizada.

El diseño de la clase política permite la introducción o eliminación de operadores, conjuntos, variables y modificadores.

Atributos de la clase Política:

Iden: String que contiene el identificador de la política.

Desc: String que contiene una descripción de la política.
Conjuntos: Tabla Hash de los conjuntos borrosos de la política.
Modificadores: Tabla Hash con los modificadores de la política.
Variables: Tabla Hash con las variables lingüísticas contenidas en la política.
Operadores: Tabla Hash con los operadores de la política.
Metcorr: Referencia al método de correlación incluido en la política.
Metdefuzz: Referencia al método de defusificación de la política.

Métodos de la clase Política:

Constructora Política(String iden, String desc)

Método que crea una política nueva.

Método addConjunto (Conjunto c)

Añade el conjunto borroso c a la política, si no está incluido.

Método addModificador (Hedge m)

Incluye el modificador m en la política si no ha sido incluido anteriormente.

Método addVariable(Variable v)

Método que incluye la variable v en la política, si no está ya incluida.

Método addOperador(Operador o)

Añade el operador o a la política si no está ya incluido en ella.

Método delConjunto(String id)

Borra el conjunto, identificado por id, de la política si es que está en ella.

Método delModificador(String id)

Borra el modificador, identificado por id, de la política si es que está en ella.

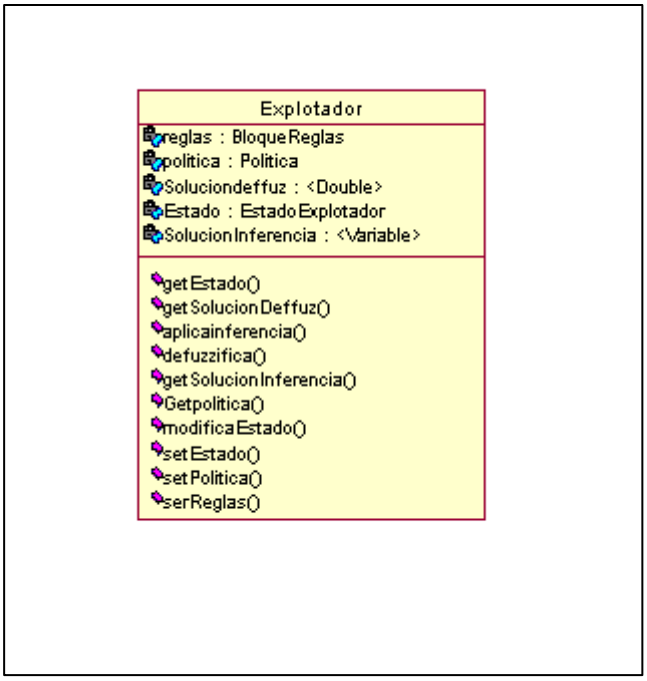
Método delVariable(String id)

Borra la variable lingüística, identificado por id, de la política si es que está en ella.

Método delOperador(String id)

Borra el operador, identificado por id, de la política si es que está en ella.

Clase Explotador



Esta clase contendrá las implementaciones que representan a los explotadores. Los explotadores son los responsables de realizar el proceso de inferencia y el proceso de defuzificación de la aplicación, a partir de los datos de entrada, estos datos serán la política a usar, la base de conocimiento y los datos de entrada.

El explotador comenzará la ejecución siempre y cuando tenga todos los datos disponibles, para controlar este aspecto se ha dotado al explotador de una variable de estado llamada EstadoExplotador que indica si todos los datos están listos y cuáles de ellos faltan, así como en qué fase de la ejecuación se encuentra en un determinado instante.

El resultado obtenido por el explotador será distinto dependiendo de la política y la base de reglas que utilice.

Atributos de la clase Explotador:

Política: Referencia a la política a utilizar por el explotador.

MetCorr: Método de correlación que utilizar el explotador.
MetDefuzz: Método de defusificación que utiliza el explotador.
Reglas: Base de reglas disponibles para el explotador.
Solución: Tabla Hash con el resultado de la inferencia.
Estado: Variable que contiene el estado en que se encuentra el explotador.

Métodos de la clase Explotador:

Constructora Explotador(Política p, MetodoCorrelacion mc, MetodoDefuzz md, BaseReglas br)

Método que construye el explotador, dando valor a todos los atributos de la clase.

Método ejecuta()

Método que realiza la inferencia.

Método deffuzifica()

Método que realiza la defuzzificación. Calcula la variable solución, y la almacena también en el estado del explotador.

Método setPolitica()

Método que establece una nueva política en el explotador.

Método setMetDefuzz(MetodoDefuzz m)

Método que establece un nuevo método de defuzzificación en el explotador.

Método setMetCorr (MetodoCorrelacion m)

Método que estable un nuevo método de correlación en el explotador.

Método setReglas(BaseReglas b)

Método que establece un nuevo conjunto de reglas en el explotador.

Metodo getMetodoDefuzz()

Método que devuelve el método de defuzzificación usado por el explotador

Método getPolitica()

Método que devuelve la política usada por el explotador.

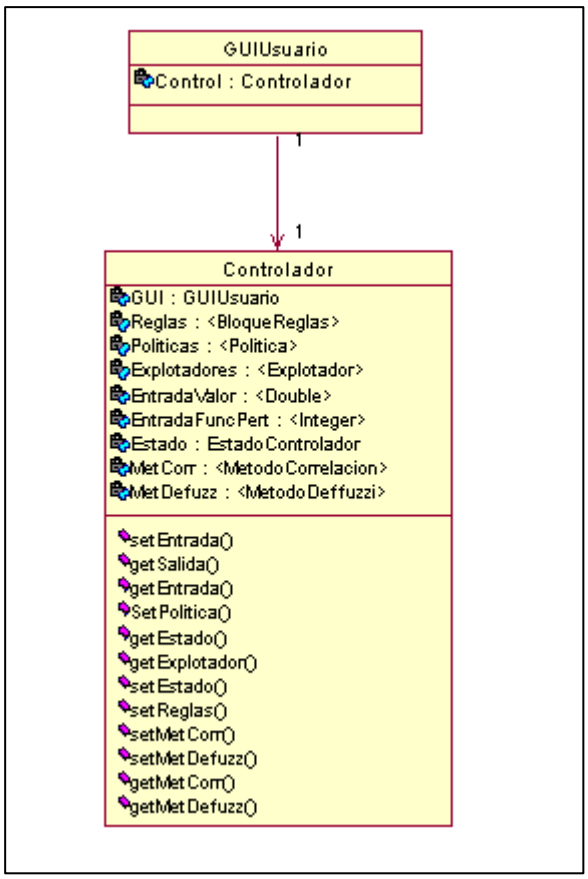
Método getReglas()

Método que devuelve el conjunto de reglas usadas por el explotador.

Método getSolución()

Método que devuelve la solución de la inferencia.

Clase Controlador



La clase controlador gestiona el proceso del control de la aplicación que estamos implementando. El controlador contiene todo el diccionario del sistema, que consta de todas las políticas, de todas las bases de reglas así como todos los métodos utilizados en los procesos de inferencia y de defuzificación.

Una vez que el controlador posee todos los datos necesarios para empezar la ejecución del sistema, éste le pasa los datos al explotador que es quien se encarga de ejecutarlo.

Al igual que en el caso del Explotador el controlador posee una variable de estado que indica si están todos los datos necesarios para realizar el proceso.

Atributos de la clase Controlador:

Reglas: Base de reglas existente en el sistema.

Políticas: Conjunto de políticas existentes en el sistema.

MetCorr: Métodos de correlación existentes en el sistema.

MetDefuzz: Métodos de defuzzificación existentes en el sistema.

Estado: variable con todos lo datos del sistema.

Métodos de la clase Controlador:

Constructora Controlador()

Constructora por defecto

Constructora Controlador(Hashtable r,Hashtable p,GUIUsuario gui, Hashtable metc,Hashtable metd)

Constructora que crea los atributos de la clase.

Método setReglas(Hastable r)

Método que establece la base de reglas del sistema.

Método setPoliticas(Hastable p)

Método que establece las políticas del sistema.

Método setMetCorr(Hastable m)

Método que proporciona un método de correlación al sistema.

Método setMetDef(Hastable m)

Método que proporciona un método de defuzzificación al sistema.

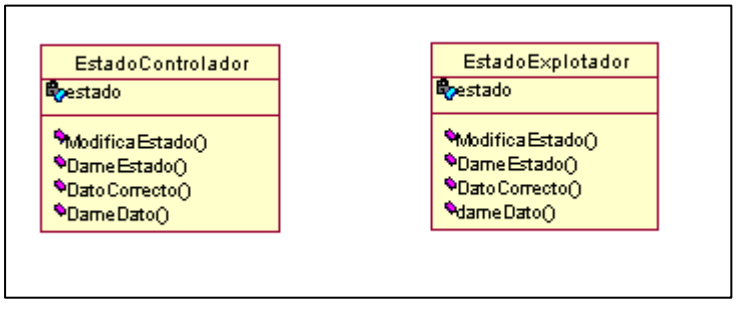
Método getReglas()

Método que devuelve la base de reglas del sistema.

Método getPoliticas()

Método que devuelve las políticas definidas en el sistema.

Clase EstadoControlador y EstadoExplotador

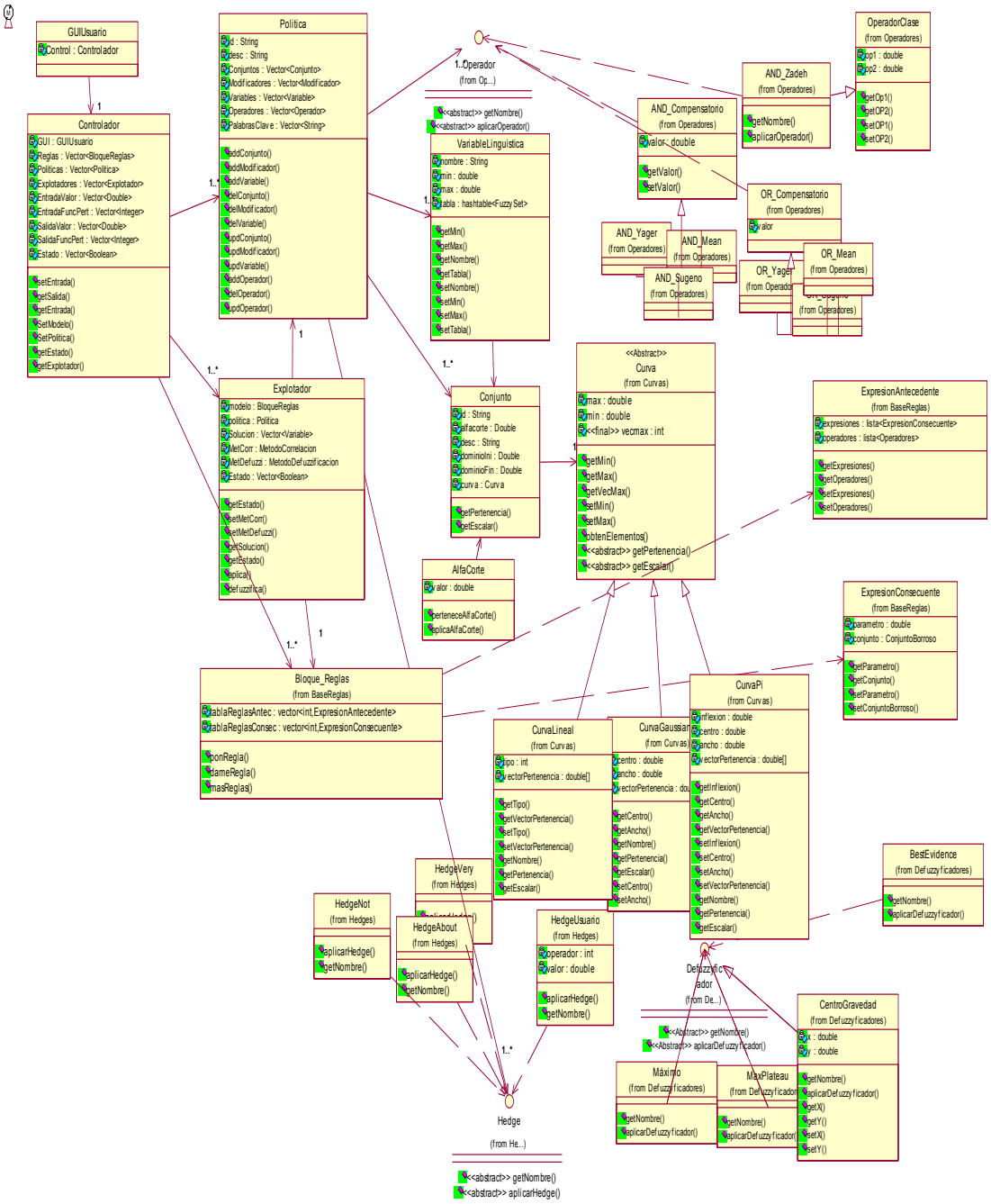


Estas dos clases implementan las variables de estado del controlador y de los explotadores. Estas variables indican si los datos necesarios para ejecutar la inferencia están listos.

Cada una de estas variables será implementada con una tabla Hash, que contendrá una referencia a los datos del sistema, tales como los operadores, la base de reglas, la política, etc...

Los métodos que contienen estas clases son las constructoras, los métodos modificadores y accesores.

Diagrama del Esquema Completo



CONCLUSIONES

Después de haber desarrollado el prototipo experimental de la herramiento Fuzzy System Editor (FuSiE), prototipo que es capaz de modelizar fácilmente sistemas basados en lógica borrosa, concluimos que:

En las pruebas realizadas a la herramienta, para los casos de uso que se describen a continuación, se determina que el diseño presenta numerosas ventajas, carentes algunas de ellas, en herramientas consultadas en la literatura al efecto.

El prototipo experimental realizado con la herramienta Java, es inherentemente multiplataforma, por ser esta una característica del lenguaje de programación utilizado para su implementación.

Este diseño aporta, mayor flexibilidad frente a las mejoras durante el período de explotación de la herramienta, rehusabilidad del sistema y mayor comprehensión de los datos, lo que evita redundancias y ambiguedades.

Las interfaces definidas en la herramienta, hacen que los subsistemas de diseño sean más fáciles de utilizar, evitando así el acoplamiento entre estos, mientras que cada subsistema implementa sus responsabilidades.

Casos de Uso

Caso de Uso: Añadir un nuevo Hedge.

Para incluir un nuevo hedge o modificador en el sistema, bastará con crear la clase que defina el objeto hedge concreto y que implemente la interfaz Hedge.

Caso de Uso: Modificar un Hedge.

Se puede modificar fácilmente la operación que describe un modificador sobre un conjunto borroso, sin tener que modificar el manejo que el sistema hace de él.

Caso de Uso: Añadir un nuevo Operador

Para incluir un nuevo operador en el sistema, bastará con crear la clase que defina el objeto operador concreto, que implemente la interfaz Operador_Interfaz y que herede de la clase Operador.

Caso de Uso: Modificar un Operador.

Se puede modificar fácilmente la operación que describe un operador concreto, sin tener que modificar el manejo que el sistema hace de él.

Caso de Uso: Añadir una Curva.

Para incluir un nuevo tipo de curva en el sistema, bastará con crear la clase que defina el objeto curva concreta y que herede de la clase abstracta Curva.

Caso de Uso: Añadir un nueva Variable Linguística y añadir un nuevo Conjunto Borroso.

Estos dos casos de uso están intimamente ligados, la definición de dos clases distintas pero relacionadas, permite hacer más intuitiva la representación del concepto, tal y como se muestra en el mundo real, una variable describe un fenómeno y un conjunto borroso su naturaleza.

Caso de Uso: Añadir una nueva Política.

Se pueden definir nuevas políticas e incluirlas en el diccionario de políticas del sistema, para estudiar el comportamiento del método de inferencia, con distintas combinaciones de variables, operadores y modificadores.

Caso de Uso: Modificar una Política existente.

La clase Política está provista de métodos, al igual que todas las clases definidas en el sistema, que modifican los elementos que definen una política concreta, pudiendo crear distintas políticas con gran facilidad.

Caso de Uso: Añadir una nueva Base de Reglas.

Se pueden añadir nuevas bases de reglas de inferencia que ejecutar sobre las variables.

Caso de Uso: Modificar una Base de Reglas.

Se pueden añadir nuevas reglas a una base de reglas ya existente.

LÍNEAS FUTURAS

Entre las posibles líneas de continuación de este trabajo destacan las siguientes:

- Creación de una gramática para la implementación de un procesador del lenguaje aplicado a la base de reglas.

- Ampliar la funcionalidad del sistema, añadiendo otros métodos de inferencia. Esto permitirá al usuario comparar para unos mismos datos de entrada, los diferentes resultados obtenidos aplicando uno u otro método de inferencia.

- Implementar una interfaz gráfica de usuario, con menús, verificación de datos, ayudas, etc., de modo que facilite al usuario el manejo de la herramienta.

BIBLIOGRAFÍA

[APR05] “Control borroso con 4D Academic”, http://www.4dhispano.com/solutions/control_borroso.html

[Bezdek,1993], “Fuzzy Models: What Are They, and Why?”.

[FORT,2005], Proyecto de Sistemas Informáticos, Univeridad Complutense de Madrid(UCM) Curso 2004/05

[FSGE04] “Fuzzy System Graphical Editor”, http://dev.oreto.inf-cr.uclm.es/www/fusyge/index.html

[Gervas05] Apuntes de clase de Pablo Gervás sobre reingeniería.

[Lee,1990] C.C. Lee, “Fuzzy Logic in Control Systems: Fuzzy Logic Controllers” , 1990.

[MRS05] “Metodología de reingeniería del software para la remodelación de aplicaciones científicas heredadas”, http://tejo.usal.es/inftec/2004/DPTOIA-IT-2004-003.pdf

[Navarro07] Apuntes del profesor Antonio Navarro sobre Modelo Orientado a Objetos y Diseño Orientado a Objetos.

[Pedrycz, 1992], W. Pedrycz, “Selected Issues of Frame of Knowledge Representation Realized by Means of Linguistic Labels”.

[Pedrycz,1993b**],** “Fuzzy Neural Networks and Neurocomputations”.

[Pedrycz, 1990], “Processing in Relation Structures: Fuzzy Relational Equations”. Fuzzy Sets.

[ROB07] Lógica borrosa y robótica,
http://campusvirtual.unex.es/cala/epistemowikia/index.php?title=L%C3%B3gica_Borrosa_y_Rob%C3%B3tica

[Sur, Omron, 1997] Sur A&C, Omron Electronics, S.A., “Lógica Fuzzy para Principiantes”.

[UGR06], Ihosvany Álvarez López, Tesis Doctoral “Aportaciones al diseño e implementación de controladores difusos: Aplicación al curado del tabaco en hoja”. Granada, Febrero, 2006.

[Zadeh,1965], L.A. Zadeh, “Fuzzy Sets and Systems”.

[Zadeh,1979], L.A. Zadeh, “Fuzzy Sets and Information Granularity”.