

Christian Nançoz

mEdit
membership function editor
for fCQL-based architecture

christian.nancoz@bluewin.ch
DIUF – Department of Informatics
University of Fribourg, Switzerland

Master thesis - February 2004

Supervisor: Prof. Dr. Andreas Meier

Abstract

The human-oriented fuzzy Classification Query Language (*fCQL*) allows non-expert users to query relational databases in a more natural language using linguistic variables and terms instead of working with the values of the attributes. Each linguistic variable is related to an attribute and a set of terms. Terms are human notions, therefore they are subjective and fuzzy. So each term needs to be defined clearly by a membership function. Linguistic variables, terms and membership functions are stored in tables of the database, called meta-tables. The advantage of this approach is that no change has to be made on the raw data.

fCQL represents a higher layer language based on *SQL*. It does not aim to formulate general queries but specially classification queries which are then translated into *SQL*. A *fCQL* query classifies a set of records into one or several classes specified in the query. Each record is then labelled with a membership value representing its matching degree to the class it has been assigned to. Because the terms are fuzzily defined, a record does not necessary belong to an unique class but perhaps to several classes.

The membership functions have an essential role in *fCQL*. The topic of this work is therefore the development of a tool allowing the definition, the edition, the storage and the retrieval of those functions. This paper presents some aspects which led to the membership function editor called *mEdit* for a *fCQL*-based architecture.

Keywords: Fuzzy Logic, Classification, *fCQL*, membership function, relational database, data mining.

Contents

1	Introduction	1
1.1.	Motivation	1
1.2.	Goals	1
1.3.	Work's guidelines	2
1.4.	Typographic conventions	3
2	About <i>fCQL</i>	4
2.1.	<i>fCQL</i> theoretical foundations	4
2.1.1.	Fuzzy sets	4
2.1.2.	Context model	5
2.1.3.	Fuzzy classification	7
2.2.	Definition	10
2.3.	<i>fCQL</i> / <i>SQL</i> example	11
2.4.	Advantages	14
2.5.	Fuzziness application fields	15
2.6.	Related projects	15
2.6.1.	Softwares	15
2.6.2.	Models and languages	16
3	Membership functions' shapes	18
3.1.	General remarks	18
3.1.1.	Concept of composing function	18
3.1.2.	About discrete and continuous attributes	19
3.2.	Choice of shapes	19
3.3.	Shapes available in <i>mEdit</i>	20
3.3.1.	Common parameters of the <i>mEdit</i> functions	21
3.3.2.	Linear function	21
3.3.3.	S-shaped function	21
4	Membership functions' storage	24
4.1.	Membership function's format	24
4.2.	Structure of the meta-tables	25

CONTENTS

5	<i>mEdit</i> architecture	29
5.1.	Overview	29
5.2.	Applet-Servlet communication	30
5.3.	Servlet-Database communication	31
5.4.	<i>mEdit</i> activity diagram	32
6	<i>mEdit</i> Java library	35
6.1.	The <i>mEdit</i> package	35
6.1.1.	mEdit	35
6.1.2.	mEdit.client	37
6.1.3.	mEdit.database	39
6.1.4.	mEdit.exceptions	39
6.1.5.	mEdit.functions	39
6.1.6.	mEdit.util	41
6.2.	The <i>edu.rit</i> package	41
6.3.	The <i>com.incors</i> package	43
7	Using <i>mEdit</i>	44
7.1.	Software requirements	44
7.1.1.	Java Runtime Environment	44
7.1.2.	Servlets container	44
7.1.3.	Web server	45
7.2.	Installation	45
7.2.1.	Installing the meta-tables	46
7.2.2.	Installing the servlet	46
7.2.3.	Installing the applet	46
7.3.	Tutorial	47
7.3.1.	Choosing a linguistic variable and a term to work on	47
7.3.2.	Defining categories	49
7.3.3.	Defining a membership function	50
7.3.4.	Getting the confirmation	53
7.3.5.	Quitting <i>mEdit</i>	55
7.4.	For developers	55
7.4.1.	Source code documentation	55
7.4.2.	Adding a new curve's type in <i>mEdit</i>	56
7.4.3.	Using <i>mEdit</i> to access a membership value (for the <i>fCQL</i> interpreter)	57
8	Conclusion	58
8.1.	Strength of <i>mEdit</i>	58
8.2.	Improvements	59
A	<i>fCQL</i> syntax	63

CONTENTS

B	<i>SQL</i> commands creating the meta-tables	64
C	<i>mEdit</i> installation CD-ROM	66

List of Figures

2.1	Membership functions for crisp and fuzzy sets	5
2.2	Providers example with contexts	6
2.3	classification of providers after merging	7
2.4	Graphical representation of the classification space	7
2.5	Using linguistic variables and terms in the context model	8
2.6	Context-based classification using fuzzy sets	9
2.7	<i>fCQL</i> as intermediate level between the RDBMS and the user	10
2.8	Classification queries with <i>SQL</i> and <i>fCQL</i>	11
2.9	The two linguistic variables and their corresponding set of terms	12
2.10	Membership function for the term High related to the linguistic variable PurchaseAmount	12
2.11	Membership functions of the terms {High, Average, Low} related to the linguistic variable PurchaseAmount	13
2.12	The six classes generated by the terms	14
3.1	Three composing functions defining a membership function	18
3.2	Triangular and trapezoidal membership functions	19
3.3	Gaussian and bell-shaped membership functions	20
3.4	S-shaped functions with different inflexion points (ν)	23
3.5	S-shaped functions with different slopes (λ)	23
4.1	The redesigned <i>fCQL</i> meta-tables and their relations	26
4.2	Structure of the <i>fCQL</i> meta-tables	27
5.1	<i>mEdit</i> architecture	29
5.2	<i>mEdit</i> activity diagram	33
6.1	The <code>mEdit.jar</code> file composed by three packages	35
6.2	UML caption for the class diagrams	35
6.3	The <code>mEdit</code> package	36
6.4	The <code>mEdit.client</code> Java package	38
6.5	The <code>mEdit.database</code> Java package	39
6.6	The <code>mEdit.exceptions</code> Java package	39
6.7	The <code>mEdit.functions</code> Java package	40

LIST OF FIGURES

6.8	The <code>mEdit.util</code> Java package	41
6.9	The <code>edu.rit.numeric</code> Java package	42
6.10	The <code>edu.rit.numeric.plot</code> Java package	43
7.1	The <i>mEdit</i> installation package	45
7.2	The <code>database.properties</code> configuration file	47
7.3	The settings window - first window appearing when starting the applet	48
7.4	The two scrolling lists for the linguistic variables and their related terms	48
7.5	Defining the categories and their membership value	49
7.6	Defining a membership function	51
7.7	Error message when a membership function is not fully defined	52
7.8	The tool tip texts for a simple use of <i>mEdit</i>	53
7.9	The two kinds of positive confirmations	54
7.10	The two kinds of negative confirmations	54
7.11	The final window	55

Introduction

1.1. Motivation

After having developed an interpreter for the fuzzy classification query language, named *fCQL*, the Information Systems Research Group of the University of Fribourg (Switzerland) faced new issues to improve its functionality.

An important drawback of this interpreter was the way it handled the membership values. Originally it was limited to discrete (or categorical) attributes so that a membership value had to be defined for each possible attribute's value. Using a discrete membership function is only possible for discrete attributes. It is not acceptable to assume that every attribute in a database is categorical and furthermore not possible to assign manually a membership value to each continuous attribute's value. For being able to use this interpreter in real business situations an extension for continuous attributes was required.

When using this interpreter, another problem was the fact that the whole membership values were predefined in the installation script and that modification of those values were only possible through *SQL* requests. That's why the environment of *fCQL* needed an editor to define and edit those membership functions. All these enhancements are brought by the membership function editor called *mEdit*.

1.2. Goals

The main task of this work was to provide a software able to define, to edit, to store and to retrieve membership functions from a relational database in order to make them available within the *fCQL* interpreter for real business applications.

To support both categorical and continuous attributes, two different handling had to be implemented according to the attribute's type. For categorical attributes, the user defines an accurate membership value for each category whereas for continuous attributes, *mEdit* allows the user to directly draw a membership function on the whole definition domain of the attribute.

The main part of this work was to design a graphical function editor and to

1.3 Work's guidelines

find a way of storing these defined functions. *mEdit* had to offer the user several appropriate shapes (each shape concealing an equation) for drawing the membership functions. On the one hand, a graphical interface should assist the user in defining the membership function. On the other hand, the equation of the membership function should be hidden from the user and had to be stored as meta-information in the meta-tables for the interpreter. Consequently the editor had to provide the *fCQL* interpreter an access point to the membership functions and values.

Additionally some software design requirements had to be respected. The robustness of the *mEdit* had to be hardly tested and the software had to be extensible to allow the definition of new membership function's shapes. Moreover this tool had to be easily used by non-statisticians. Therefore no equation had to be entered by the user as the graphical user interface should allow to define a function in a simple and graphical way.

1.3. Work's guidelines

This report follows the development steps of *mEdit*. First of all, chapter 2 presents an overview of *fCQL* to provide an intuitive approach of this language with the help of an example and the technical background required for understanding the overall context of the editor. Some main advantages of *fCQL* and different applications fields are concisely exposed. To close this chapter, the last section will briefly present other softwares related to fuzzy logic and other fuzzy languages.

From a theoretical point of view, a relevant issue of this work is the choice of the appropriate shapes and their related equations which is extremely important for defining the membership functions. Chapter 3 exposes the existing shapes according to their application fields and then presents the functions available in *mEdit*.

The next step was to consider the way the functions should be stored in the meta-tables of the database. This implied finding a format of storage for the equations and remodelling the meta-tables. The ideas related to this topic are explained in chapter 4.

Before starting the implementation of the editor, a well designed architecture had to be developed. The *mEdit* architecture is introduced in chapter 5 which also details the interaction between its different components and specifies the chosen technologies. In section 5.4, the different activities of the editor are briefly described to understand the sequence of windows while using the editor.

1.4 Typographic conventions

Chapter 6 gives information about the editor's library implemented in Java. Every subpackage is accurately described and the role of the most important classes is explained. The library and its different packages are also illustrated by class diagrams.

Chapter 7 provides the installation requirements and the instructions for an optimal usage of *mEdit*. The last section presents some tips for developers interested in future extensions and provides some general instructions for modifying *mEdit*.

Finally, some words about the advantages and the potential improvements of *mEdit* conclude this report in chapter 8. As annexes, the *fCQL* syntax and the *SQL* statements to create the meta-tables end this paper.

1.4. Typographic conventions

To clarify the reading of this work, some typographic conventions will be applied. The name of directories, files, extensions, programming languages, softwares and trademarks will be printed in *italic characters*. Samples of code source, classes, methods and packages will be printed in *typewriter's characters*. Additionally important expressions will be printed in **bold characters** the first time they appear in the text.

About *fCQL*

2.1. *fCQL* theoretical foundations

The theoretical basis of *fCQL*, for fuzzy Classification Query Language, finds its roots in the fuzzy sets theory from L.A. Zadeh¹ and the context model. The context model allows the classification of database records and the fuzzy sets theory makes the classification fuzzy.

2.1.1. Fuzzy sets

The notion of fuzzy set stems from the work of Zadeh [28] published in 1965. Zadeh observed the gap between mental representations of the reality, that is human concepts, and usual mathematical approaches. Human concepts are represented by natural language terms like "young man" or "high price". Such concepts are useful to describe the reality and to summarize the human perception of the world. However, they are vague and subjective. Since each person has its own idea of the meaning of an expression such as "young man", human concepts are therefore subjective and context-dependant. On the other hand, mathematical concepts are sharp and sometimes difficult to understand. Therefore sharp representations are not adequate to describe the vagueness of the human perception.

As natural language terms are vague, a gradual notion is needed to define those terms. The way of defining mental representations is no more "all-or-nothing" but is expressed by the notion of **membership** which is represented by a value, called **membership value**, in the interval $[0, 1]$. The specificity of fuzzy sets is to capture this idea of partial membership whereas classical crisp sets are reduced to binary membership, in other words $\{0, 1\}$. Every member of a crisp set has a membership value of 1 and the outside elements have a membership degree of 0. Therefore, a crisp set does not need to label each element with a membership value. In contrast, every member of a fuzzy set has an associated membership value. For instance, a fuzzy set can enclose a specific element X with a membership value of 0.6. This same element X can also be part of one

¹ Professor at the Computer Science Division Department of Electrical Engineering and Computer Sciences University of California (see <http://www.cs.berkeley.edu/~zadeh/>)

or more other fuzzy sets with different membership degrees. Thus the notion of membership allows an element to be part of several fuzzy sets.

Each fuzzy set is represented by a function called **membership function**. A membership function associates a membership value with each element in the referential of the related fuzzy set. Therefore, a fuzzy set is often understood as a membership function. As figure 2.1 shows, a membership function can define the set of all the "young men". If a crisp set is used, only the men between 20 and 36 years of age are considered to be young men. On the other hand, the fuzzy set is not so precise: the men between 24 and 32 are "young" with a membership degree of 1, but the one between 16 and 24 and between 32 and 40 are also "young men", however with partial membership degrees.

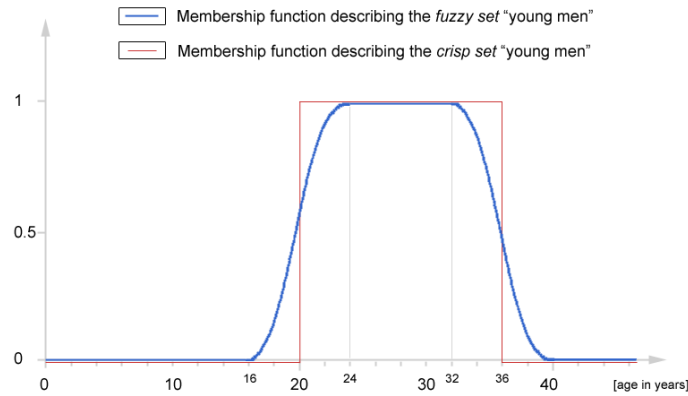


Figure 2.1: Membership functions for crisp and fuzzy sets

To extend the functionalities of fuzzy sets, operators of the classical set theory like intersection, union, complement, inclusion and so on have been adapted to the concept of fuzzy sets. For instance, the union of two fuzzy sets can be performed by taking the pointwise maximum of their membership functions and their intersection by the pointwise minimum of their membership functions. It is important to note that, due to the new potential of the fuzzy sets theory, these operations can also be defined in different ways.

2.1.2. Context model

In the relational data model, every attribute A_j is defined on a domain $D(A_j)$. The enhancement of the context model consists in a context $K(A_j)$ assigned to each attribute A_j . A context $K(A_j)$ is a partition of $D(A_j)$ into equivalence classes. Thus a relational database schema with contexts $R(A, K)$ is composed by a set $A = (A_1, \dots, A_n)$ of attributes and a set $K = (K(A_1), \dots, K(A_n))$ of associated

2.1 *fCQL* theoretical foundations

contexts. If each equivalence class contains only one element, the context model has the same effect as the relational data model.

Shenoi [26] introduced the context-based fuzzy equivalence which generalizes the classical equality and enhances the notion of tuples redundancy found in the relational model. In contrast to the relational model, two context-redundant tuples are not identical but equivalent. Two tuples t and t' are context-redundant if for each component t_i of t , the corresponding component t'_i of t' belongs to the same equivalence class. This means that all the context-redundant tuples are part of a same class. Therefore the context model leads to a classification.

Figure 2.2 shows an example of contexts for a database table containing three columns: the provider's name, the quality of service and the delay in delivering the service. The idea is to use the context model for classifying the providers according to their quality and delay.

$$\begin{aligned}
 D(\text{Provider}) &= \{ \text{Dewag}, \text{MAM}, \text{KBA}, \text{MTX} \} \\
 D(\text{Quality}) &= \{ \text{low}, \text{sufficient}, \text{average}, \text{high} \} \\
 D(\text{Delay}) &= [1, 10] \\
 \\
 K(\text{Provider}) &= \{ \{ \text{Dewag}, \text{MAM}, \text{KBA}, \text{MTX} \} \} \\
 K(\text{Quality}) &= \{ \{ \text{low}, \text{sufficient} \}, \{ \text{average}, \text{high} \} \} \\
 K(\text{Delay}) &= \{ [1, 5], [6, 10] \}
 \end{aligned}$$

<u>Provider</u>	Quality	Delay	tuple
Dewag	low	3	t_1
MAM	average	7	t_2
KBA	sufficient	5	t_3
MTX	high	2	t_4

Figure 2.2: Providers example with contexts

The contexts have been defined on the three attributes "Quality", "Delay" and "Provider". The quality values "low" and "sufficient" are equivalent and thus they are in the same equivalence class. The same applies to the quality values "average" and "high", as well as the delay intervals $[1, 5]$ and $[6, 10]$. In contrast, all the providers are included in a same and unique equivalence class corresponding to the referential of the "Provider" attribute.

Once the context-redundant tuples are detected, a merge operation on each

group of context-redundant tuples creates the resulting classification. Figure 2.3 shows this classification for the providers example, after the merge operation. All the attribute's values of tuples t_1 and t_3 belong to the same equivalence classes. Indeed, the providers "Dewag" and "KBA" are in the same equivalence class and the qualities "low" and "sufficient" are equivalent as well as the delays "3" and "5". Therefore the two tuples are context-redundant and belong to the same class called "C4" as shown in figure 2.4. Figure 2.4 graphically represents the four different classes in the classification space defined by the attributes "Delay" and "Quality" and generated by those contexts.

Provider	Quality	Delay
{Dewag, KBA}	{low, sufficient}	{3, 5}
{MAM}	{average}	{7}
{MTX}	{high}	{2}

Figure 2.3: classification of providers after merging

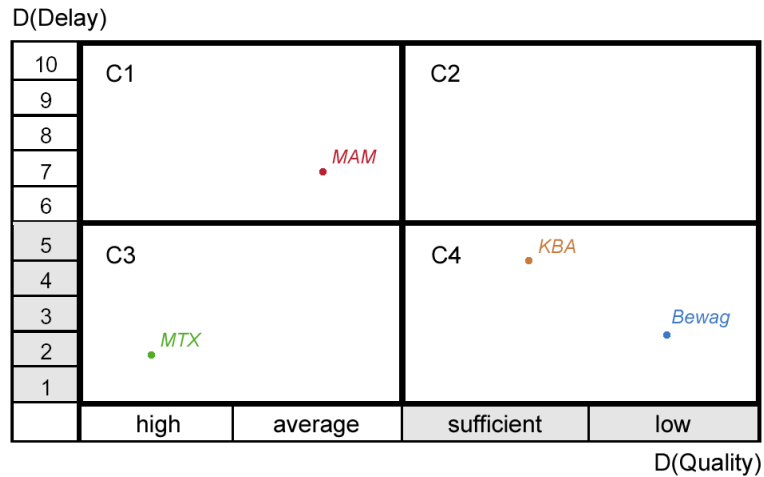


Figure 2.4: Graphical representation of the classification space

2.1.3. Fuzzy classification

As explained in the previous subsections, the fuzzy sets theory allows partial memberships to sets. On the other hand, a class can be seen as a fuzzy set. The main

idea of the fuzzy classification [10] [26] is to consider equivalence classes generated by the context model as fuzzy sets. To create fuzzy classes, a verbal term is assigned to each class. Since verbal terms are vague, the classes become vague too. The notion of **linguistic variable** introduced by Zimmermann [30] is useful to perform this mapping. By definition, a linguistic variable takes a value in a set of verbal terms. As shown in figure 2.5, the linguistic variable "Delay" can take one of the "acceptable" or "unacceptable" term's values. Each linguistic variable is related to an attribute of a database table. In the example of figure 2.5, the linguistic variable "Delay" is related to the attribute with the same name. Often, for the sake of simplicity, the name of the linguistic variable is the same as its related attribute. Then the terms "acceptable" and "unacceptable" represent each an equivalence class defined on the domain of the related attribute.

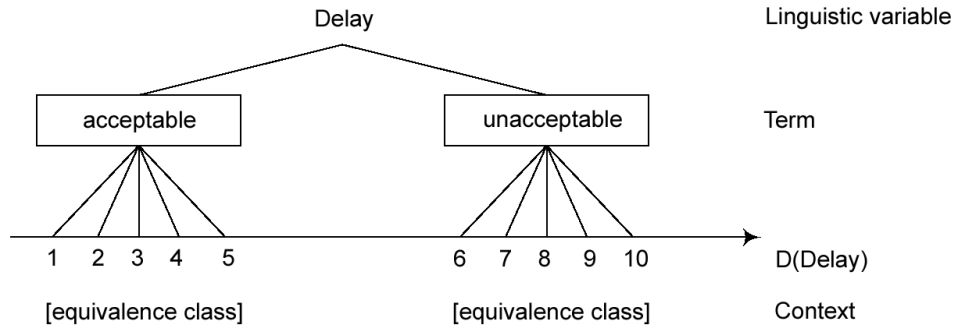


Figure 2.5: Using linguistic variables and terms in the context model

Since terms can be seen as fuzzy sets defined by a membership function, the membership to an equivalence class is no more binary like in the context model but can be partial. The tuples can be part of several classes with different membership degrees. Based on the providers example of the previous subsection, figure 2.6 shows the four classes using terms and membership functions. Each class can have its own semantic. For instance, the class of providers "C4" could be labeled "improve quality" as the quality is bad. The class "C2" could represent the providers whose relationship has to be reconsidered because of their bad quality and unacceptable delay.

The two linguistic variables are "Quality" and "Delay" and their respective set of terms are {good, bad} and {acceptable, unacceptable}. The terms "good" and "bad" are described by their membership function, respectively μ_{good} and μ_{bad} . Those two functions partition the domain $D(Quality)$ in two fuzzy sets. The membership of the provider "MAM" (quality "average" and delay "7") to the fuzzy set "good" is given by the following formula:

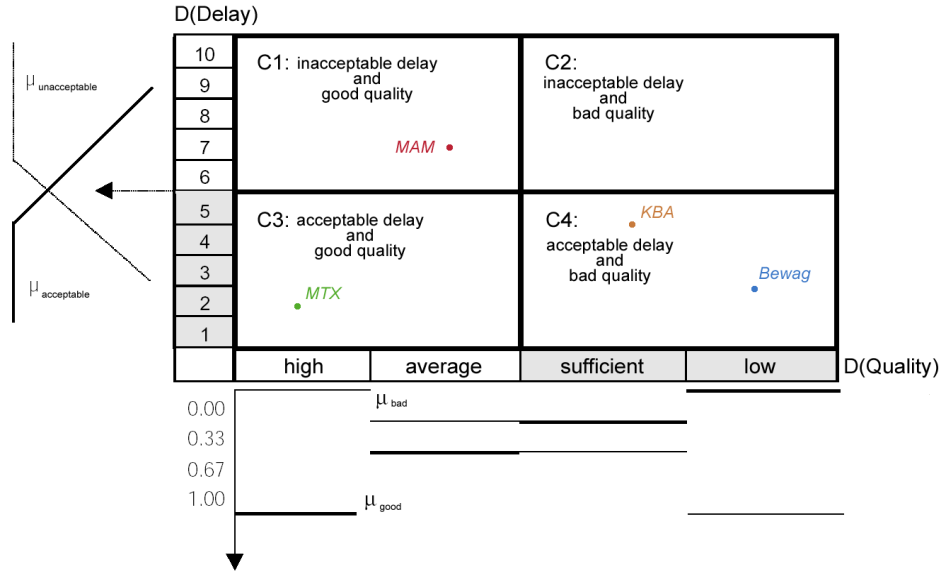


Figure 2.6: Context-based classification using fuzzy sets

$$M(MAM|good) = \mu_{good}(average) = 0.67$$

In a same way, its membership to the fuzzy set "bad" is given by:

$$M(MAM|bad) = \mu_{bad}(average) = 0.33$$

The membership value of an element to a specific class is defined by the aggregation of all the terms of linguistic variables defining the class. In the providers example, the four classes are defined by the two "Quality" and "Delay" linguistic variables. Since the class "C4" is defined by an "acceptable" delay and a "bad" quality, the membership degree to the fuzzy set "acceptable" and the one to the fuzzy set "bad" have to be aggregated. Depending on the operator performing the aggregation, more importance is given to one or several particular linguistic variables. The operator chosen in *fCQL*, called gamma operator, allows an effect of compensation between the different linguistic variables. The following formula shows how to calculate the aggregated membership of the provider "MAM" to the class "C4":

$$M(MAM(average, 7)|C4) = f_{aggregation}(\mu_{bad}(average), \mu_{acceptable}(7))$$

2.2. Definition

Originally created by Schindler [10], *fCQL* is a fuzzy Classification Query Language founded on the fuzzy classification approach presented in the previous section. This language allows the user to formulate classification queries on a linguistic level without numerical values, but with predefined linguistic variables and verbal terms. As figure 2.7 shows, *fCQL* is a higher level language based on *SQL* and aims to use verbal terms to classify the records of a relational database. The *fCQL* queries are processed by an additional layer built on the RDBMS², the *fCQL* interpreter. This component translates the *fCQL* queries into *SQL* queries sent to the RDBMS.

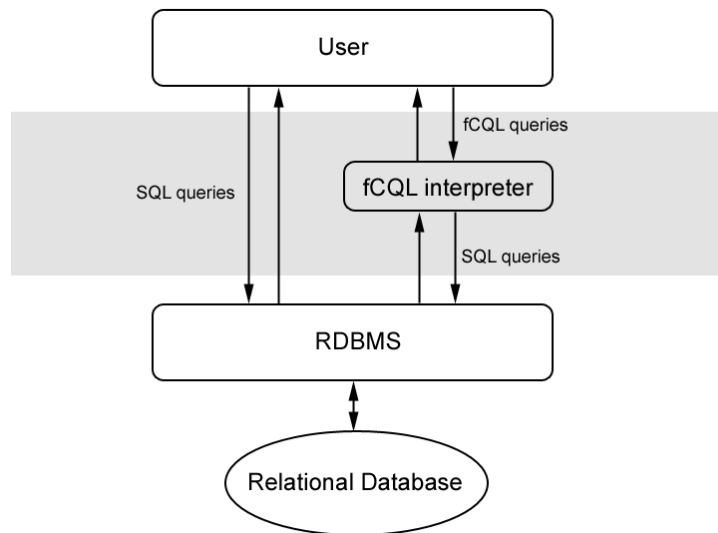


Figure 2.7: *fCQL* as intermediate level between the RDBMS and the user

Unlike the widely used *SQL* language, *fCQL* aims to build only fuzzy classification queries. It is also possible with *SQL* to formulate classification queries by selecting the records matching specific criteria. However, the classification made by *SQL* is sharp and does not include a membership degree for each matching record like *fCQL* does. Actually, after running a *fCQL* query, the *fCQL* interpreter returns the records and their degree of membership to the class or classes formulated in the query. Figure 2.8 shows an example of how *SQL* can be used to classify records and the corresponding *fCQL* query. As the *WHERE*-clause of a *SQL* query which is the selection condition containing one or several attributes, the *WITH*-clause of *fCQL* is the classification predicate using linguistic variables and terms. These

² Relational Database Management System

2.3 *fCQL* / *SQL* example

queries will be explained in details in the next section and the complete *fCQL* syntax is available in appendix A.

$$\begin{array}{l} SQL \left\{ \begin{array}{l} \text{SELECT CustomerID FROM customers} \\ \text{WHERE PurchaseAmount} \geq x \\ \text{AND SeminarPresence} \geq y \end{array} \right. \\ \\ fCQL \left\{ \begin{array}{l} \text{CLASSIFY CustomerID FROM customers} \\ \text{WITH PurchaseAmount IS High} \\ \text{AND SeminarPresence IS Sufficient} \end{array} \right. \end{array}$$

Figure 2.8: Classification queries with *SQL* and *fCQL*

Since the *fCQL* queries use predefined linguistic variables and terms, the *fCQL* interpreter lays the definitions of this useful meta-information in the same relational database which already contains the raw data. Independent tables called **meta-tables** are added to the relational database schema for storing those definitions, so no modification or migration of the raw data is required.

2.3. *fCQL* / *SQL* example

To show the potential of *fCQL*, we assume that a business relational database contains a table called `customers` with information about the customers with three attributes: `CustomerID`, `PurchaseAmount` which is the total amount of purchases bought by the customer and `SeminarPresence` which is the number of frequented seminars organized by the seller and which quantifies in a certain way the client's interest for the products.

If the marketing department wants to know who are among all the valuable customers, an approach would be to use *SQL* to find out which customers belong to this class. The only available criterion to define the class of valuable customers is the information contained in the database. The generic query to retrieve the valuable customers is expressed in figure 2.8. An issue is to determine the x and y values. Assume a valuable customer should have a minimal purchase amount of 10'000 CHF and a minimal number of frequented seminars of 5. The drawback of such *SQL* query for the classification of customers is its sharpness: a customer with a purchase amount of 100'000 CHF and a number of frequented seminars of 4, shortly written [100'000, 4], would be excluded of the result set. It turns out that such a customer could be as valuable as a [10'000, 5] customer. If the businessman had to choose himself the valuable customers, he would surely

2.3 *fCQL* / *SQL* example

choose the [100'000, 4] customer and perhaps not the [10'000, 5] one.

To avoid the skewed classification that a *SQL* request may generate, the *fCQL* syntax proposes to formulate a human-oriented query as shown in figure 2.8. In that case, the two linguistic variables are `PurchaseAmount` and `SeminarPresence`, which have the same name as their related attribute, and take a value in their respective set of terms, {`High`, `Average`, `Low`} and {`Sufficient`, `Insufficient`}. This situation is illustrated in figure 2.9.

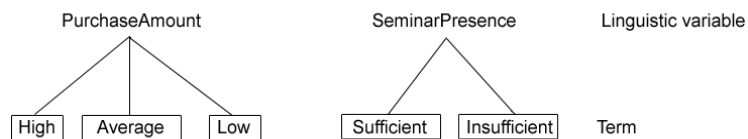


Figure 2.9: The two linguistic variables and their corresponding set of terms

In a first step, only the linguistic variable `PurchaseAmount` will be considered. This variable being associated to the attribute with the same name, all the terms of this variable have their membership function defined on the domain of the related attribute. Concretely, the term `High` has a membership function which links a membership value to each value that can take the amount of purchases. If a customer has a membership value of 1 regarding to the amount of purchases, it means that his amount fully corresponds to the concept `High`. If this value is 0.5, the amount partially matches the definition of `High`. If the matching degree is 0, the amount does not correspond to that concept at all. Figure 2.10 shows a possible membership function associated to the term `High`.

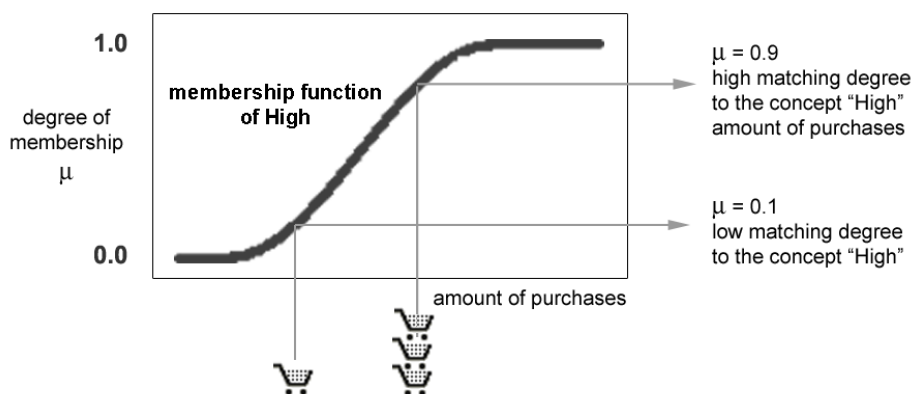


Figure 2.10: Membership function for the term `High` related to the linguistic variable `PurchaseAmount`

2.3 *fCQL* / *SQL* example

Since the notion of `High` is defined by a membership function, the answer to the question "Does this customer have a high amount?" is no more yes or no. Assume that `A1` is the class of all the customers with a high amount of purchases and `A2` the class for low amounts. If the membership function of `High` returns 0.7 for a given customer, the fuzzy answer to the above question is "70% yes and 30% no". Moreover this means that this customer does not only belong to the class `A1` but also to the class `A2`. In a similar way, the membership function of the term `Low` can be used to compute the membership to the class `A2`. Figure 2.11 gives an example of triangular and trapezoidal membership functions for the three terms `{High, Average, Low}`.

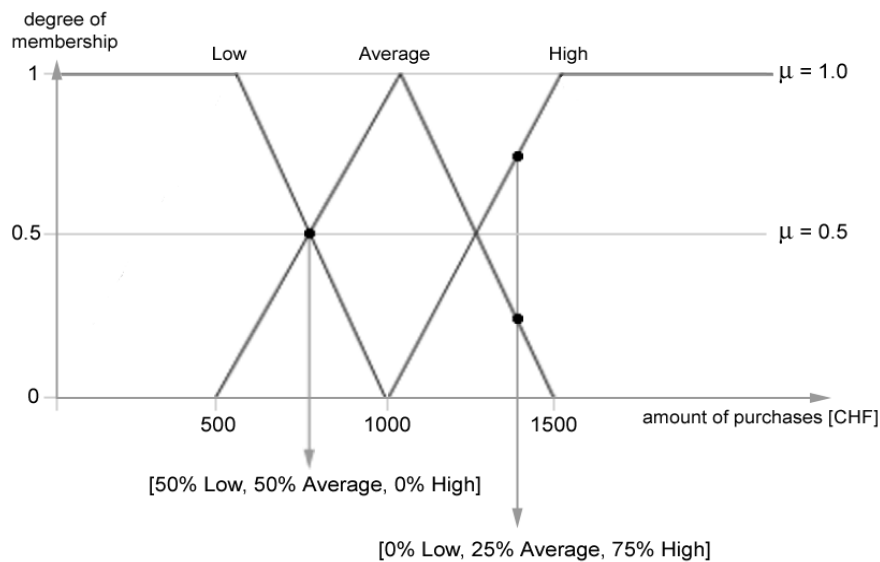


Figure 2.11: Membership functions of the terms `{High, Average, Low}` related to the linguistic variable `PurchaseAmount`

But that is not yet what the business user wants to know. With the same process, a membership function can be defined for the term `Sufficient` related to the linguistic variable `SeminarPresence`. As shown in figure 2.12, assuming that `C1` is the class of all the customers with a high amount of purchases and a sufficient number of presences at the seminars, an aggregation function can be applied on these two membership values (the membership value for `High` and the one for `Sufficient`) to calculate a new membership value: the membership degree to the class `C1`. As the gamma operator used in the *fCQL* interpreter allows an effect of compensation between the linguistic variables, the operator can be configured to compensate a low membership value for `Sufficient` if the membership value for `High` is very high. Such effect is useful to give more importance to a specific

2.4 Advantages

linguistic variable.

PurchaseAmount	High	C1	C2
	Average	C3	C4
	Low	C5	C6
		Sufficient	Insufficient
			SeminarPresence

Figure 2.12: The six classes generated by the terms

With this type of query using terms defined by their associated membership function, it is possible to keep customers who would have been rejected by a sharp classification done by a *SQL* query. Indeed, some customers who are valuable for the user are surely rejected by the *SQL* query but can be retrieved by a *fcQL* query if the membership functions are well defined. Therefore, *fcQL* avoids rejecting potential valuable customers by using simple verbal terms and simulates the human thinking when classifying the customers.

2.4. Advantages

As introduced in the previous sections, *fcQL* allows the user to formulate intuitive and human-friendly queries to make a fuzzy classification. With the help of linguistic variables and terms representing human concepts, the user creates comprehensible and simple requests. Moreover the complexity of the membership functions is hidden by the terms and is not visible to the user. Indeed, the membership functions are defined by business specialists and the user simply uses the predefined concepts.

The main advantage of the fuzzy classification is that each matching record has a membership degree in all the classes it belongs. Then the user is free to interpret those membership values and to fix a acceptance threshold. This important additional information, which would not have been available without

2.5 Fuzziness application fields

fuzzy classes, allows the user to take the correct decision regarding the customers.

As the classification is one of the approaches used in data mining, this query language has to be seen as a powerful and easy to use data mining tool. It is accessible to specialists but also to non-specialist users which can plainly build user-friendly queries thanks to the simple *fCQL* syntax. Unlike *SQL*, the *fCQL* language is not reserved to database experts.

Another important advantage of this data mining tool is the number of modifications necessary to make a relational database *fCQL*-compatible. Only the meta-tables have to be installed to use *fCQL*, without changes on the raw data. This meta-information is stored beside the raw data and can be easily edited, retrieved or even deleted.

2.5. Fuzziness application fields

More widely used and known, fuzzy control was the first important usage of the fuzzy theory. Fuzzy logic could also be used in a lot of other fields, but is, in fact, only deployed in some of them. Domains like search engines for improving the precision of search results, segmentation of magnetic resonance images, search in multimedia and geographical images databases are for instance already under influence of fuzziness.

For a business usage, fuzzy logic is still rarely used and companies prefer to invest in classical data mining tools to solve their problems than to trust in a slightly different and new approach.

2.6. Related projects

2.6.1. Softwares

Other fuzzy query softwares are already on the market. *Sonalysts Inc.* offers in its fuzzy system solutions package a program named *Fuzzy Query*³. In this package, a membership function editor is also available. Moreover the language used by this software to query the relational database is very similar to *fCQL*.

Other provider of fuzzy solutions, *The MathWorks* provides *Fuzzy Logic Toolbox 2.1*⁴ which extends MathLab environments for fuzzy inference systems. A membership function editor and also a lot of other tools like a rule editor and

³ <http://fuzzy.sonalysts.com/>

⁴ <http://www.mathworks.com/products/fuzzylogic/>

powerful GUI's are available.

Another software implementing a membership function editor, *FIDE*⁵, for *Fuzzy Inference Development Environment* from *Apronix*, offers a complete environment for the development of fuzzy logic-based systems and fuzzy logic tools including rules editor and a real-time code generator. It supports all phases of application development: concept, design, tuning, simulation and implementation.

For *MATLAB* environments, *FlexTool*⁶ from *CynapSys* builds computationally intelligent systems using soft computing techniques like fuzzy systems, genetic algorithms, evolutionary algorithms, neural networks, and chaos theory.

2.6.2. Models and languages

The *fCQL* language deploys fuzziness only for classification purpose. But other languages allow general queries based on fuzziness. Two conceptual data models dealing with fuzziness exist: fuzzy queries to regular database systems and formal queries to fuzzy database systems. The latter implies migrating an existing database to a new fuzzy database and this step may be very expensive in time and effort. Because this latter model is often not a good choice in real business situations, this section will only consider the first model. The simplest way to insert fuzziness is this model, which is using fuzzy queries on a classical relational database and *fCQL* is based on it. The next paragraphs introduce other languages founded on this data model.

A first enhancement of the relational domain calculus was published by Takahashi [27]. Called *Fuzzy Query Language (FQL)*, this language is able to represent all four types of fuzzy statements distinguished by the work of L.A. Zadeh [29]. This language is a theoretical basis for developing a human-oriented interface with relational databases.

Another idea for formulating fuzzy queries is *Fuzzy SQL (SQLf or FSQL)*. This language extends *SQL* to allow flexible and imprecise queries but also preserves the original operations of the classical relational algebra. Like *fCQL*, it translates first the fuzzy sequences into usual *SQL* sequences executed directly by the RDBMS. For instance, terms can be used directly in *SQL* queries but must be prefixed by a "\$" symbol to distinguish them easily. Fuzzy comparators based on the fuzzy sets theory are also available. For each selection condition, a fulfilment threshold may be established indicating that the condition must be satisfied with a minimum membership value to be considered. Additionally, fuzzy constants may be used. In [23], Bosc and Pivert published this idea of *SQL* extension. Several prototypes validating this extension were proposed by Galindo, Cubero, Pons and

⁵ <http://www.aptronix.com/fide>

⁶ <http://www.flextool.com>

2.6 Related projects

Medina in [16] but also in [24] by Zadrozny and Kacprzyk who developed their own extension called *FQUERY* in [21].

The most well-known relational model using fuzzy logic and specially *FSQL* is *GEFRED* [20, 19] which was developed at the University of Granada (Spain). Based on *GEFRED*, the *FIRST* architecture [18] proposes a RDBMS client-server scheme to build a Fuzzy Relational Database Management System (FRDBMS). A drawback of the main component of this architecture, the *FSQL* interpreter, is its apparent limitation to trapezoidal membership functions. The next chapter will highlight the advantage of *mEdit* regarding the shapes of the membership functions.

Membership functions' shapes

3.1. General remarks

Before starting this important chapter, some preliminary remarks have to be made about the membership functions and their composition.

3.1.1. Concept of composing function

The most powerful method is to define the membership function by parts on its definition domain. The definition domain of a membership function includes all the values that the related attribute can take, that is the attribute's domain. Remember that a membership function is defined for a specific term which is related to an attribute. Each part of the membership function is a function called **composing function**. Figure 3.1 shows three composing functions CF1, CF2 and CF3 which define piecewise a rather complex membership function.

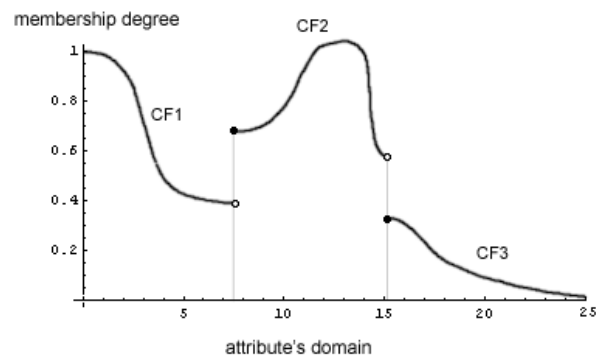


Figure 3.1: Three composing functions defining a membership function

As illustrated in figure 3.1, each composing function is defined on a specific interval of the membership function's definition domain. Between the composing functions, a vertical discontinuity may occur. However, the composing functions CF1, CF2 and CF3 do not overlap themselves and this overlapping constraint is an additional properties of composing functions. Finally, joining all the composing

3.2 Choice of shapes

functions must ensure that the membership function is defined for each point of its definition domain. Therefore, horizontal gaps are not permitted.

3.1.2. About discrete and continuous attributes

Conceptually the terms of both discrete and continuous attributes may be defined by a membership function to determine their membership values. The membership function will simply be discrete or continuous. In the implementation, the way *mEdit* deals with those two types of attribute is different. For discrete attributes, the user only defines the discrete membership values of each category. On the other hand, for continuous attributes, the user can graphically define the membership functions. Therefore in the following sections, "membership function" has to be understood as "continuous membership function".

3.2. Choice of shapes

The shape of the membership function is essential. Subsection 2.6.2 explained that the *FSQL* interpreter only uses trapezoidal membership functions. It is very usual to find trapezoidal and triangular membership functions in fuzzy logic. This kind of shapes is often used for fuzzy control of machines or robots. Figure 3.2 shows an example of such membership functions.

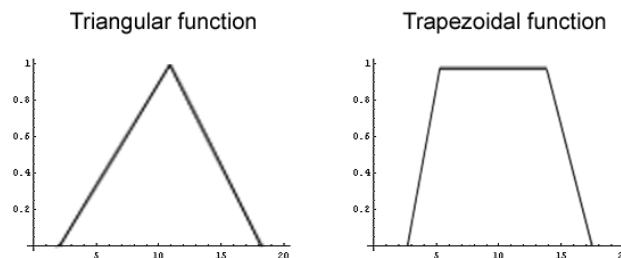


Figure 3.2: Triangular and trapezoidal membership functions

Besides trapezoidal functions, other shapes inspired by probabilistic distributions are also important. Examples of gaussian and bell-shaped functions are represented in figure 3.3. Those functions often have asymptotic properties and are based on exponential or hyperbolic factors.

3.3 Shapes available in *mEdit*

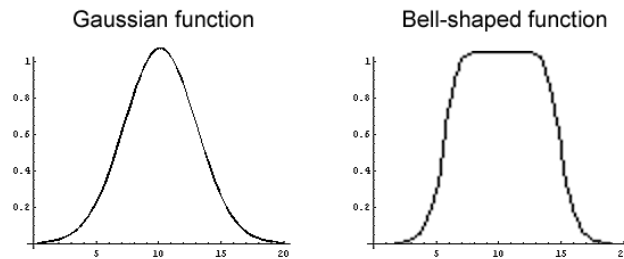


Figure 3.3: Gaussian and bell-shaped membership functions

The trouble with collecting information about the shape of membership functions is that most articles dealing with fuzzy sets do not describe the used membership functions. In [15], J. Dombi classified some found membership functions according to their nature and context:

- Membership functions based on heuristics
- Membership functions based on reliability concerns with respect to a particular problem
- Membership functions base on more theoretical demand like for instance decision making
- Membership functions for control
- Membership functions as a model for human concepts

This latter group of membership functions is the one which is interesting for the *fcQL* classification because the membership function should express what a human being means when using a specific term. In its work, J. Dombi tried to extract the common properties of all those types of membership functions and developed his own membership function presented in the next section.

3.3. Shapes available in *mEdit*

The *mEdit* user cannot foresee the exact shape of a membership function. Like in every data mining effort, the training phase of the chosen model will evaluate if the shape is appropriate or has to be adapted. Furthermore there is no standard function which models human concepts and their hidden meaning. That's why the *mEdit* editor offers two kinds of functions which are able to generate a wide range of membership functions.

3.3.1. Common parameters of the *mEdit* functions

The two types of function used in *mEdit* have a certain number of common parameters. Both functions have:

- a definition domain composed by lower and higher bounds.
- a start y-coordinate, that is the start membership value of the function and
- an end y-coordinate, that is the end membership value of the function.

3.3.2. Linear function

Linear functions lines allow to define piecewise all the following derived functions:

- horizontal functions
- triangular functions
- and trapezoidal functions.

Linear functions lines do not need more parameters than the common parameters of the *mEdit* functions. These parameters are present in the standard cartesian line equation which the *mEdit* editor uses to code each linear shape:

$$\mu(x) = \underbrace{\frac{\mu_2 - \mu_1}{b - a}}_{\text{slope}} x + \underbrace{\mu_1 - \frac{\mu_2 - \mu_1}{b - a} a}_{\text{y-intercept}}, \quad x \in [a, b]$$

where:

- a is the start x-coordinate, that is the lower bound of the definition domain of the function
- b is the end x-coordinate, that is the higher bound of the definition domain of the function
- μ_1 is the start y-coordinate, that is the start membership value of the function
- μ_2 is the end y -coordinate, that is the end membership value of the function.

3.3.3. S-shaped function

Once linear functions can be drawn, all the remaining functions are the curved shapes. The equations of those curved shapes have to respect three properties:

1. the parameters must be meaningful. The *mEdit* user must understand the direct effect of changing the parameter's value. A lot of exponential and hyperbolic functions do not have simply understandable parameters for non-statisticians.

3.3 Shapes available in *mEdit*

2. the number of parameters must be as small as possible. The more parameters there are, the more complex the configuration of the function is.
3. if the user wants the function to be traced from a start point to an end point, the equation of this function must provide parameters to realize this interpolation. The drawback of the majority of available functions is their asymptotic behavior. Given a generic equation, it is often impossible by simply setting the parameters to force this function to pass through two arbitrary specified points. For instance, a gaussian equation with distribution $N(0,1)$ ¹ will never be 0. If the user wants a membership value of 0 at a specific x-coordinate, the gaussian function will not be able to reach that value. A discontinuity will be necessary to set the value 0 at the specific x-coordinate.

J.Dombi [15] computed this ideally customizable S-shaped function which also gathers some common mathematical properties of membership functions from several articles [15] which appear between 1985 and 1988 or which were first printed in *Fuzzy Sets and Systems*². This function respects some convexity constraints of a S-shape and defines the membership as a function of the distance $d(x)$ between a given object and its ideal standard. Furthermore, its equation is a rational function of polynomials and can be configured to get a linear form. This function is described by the following equation:

$$\mu(x) = \frac{(1-\nu)^{\lambda-1}(x-a)^\lambda}{(1-\nu)^{\lambda-1}(x-a)^\lambda + \nu^{\lambda-1}(b-x)^\lambda}, \quad x \in [a, b]$$

where:

- a is the start x-coordinate, that is the minimum of the definition domain of the function.
- b is the end x-coordinate, that is the maximum of the definition domain of the function.
- $\nu \in]0, 1[$ quantifies the x-position of the inflexion point between a and b . If ν is close to 0, respectively 1, the inflexion point is close to a , respectively b .
- λ is the slope of the function. If $\lambda = 0$, the function is a horizontal line and if $\lambda = 1$ a linear function:

$$\lambda = 1 \Rightarrow \mu(x) = \frac{x-a}{b-a};$$

For *mEdit*, the convention was made to allow λ values between 2 and 20. A slope with a λ value of 20 already approaches a vertical line.

¹ $N(0,1)$ means normal distribution with mean 0 and variance 1

² the official publication of the International Fuzzy Systems Association (IFSA)

3.3 Shapes available in *mEdit*

The function given by Dombi is normalized to draw a S-shape from $(a, 0)$ to $(b, 1)$. To allow the user to use it between two arbitrary points (a, m_1) and (b, m_2) where m_1 is the start membership value and m_2 the end membership value, the function used in *mEdit* was modified :

$$\mu_{mEdit}(x) = m_1 + \mu(x)(m_2 - m_1), \quad x \in [a, b]$$

Figures 3.5 and 3.4 illustrate some samples of S-shapes with different ν (position of inflexion point) and λ (slope) parameters to show the possibilities of the S-shaped function developed by Dombi and available in *mEdit*.

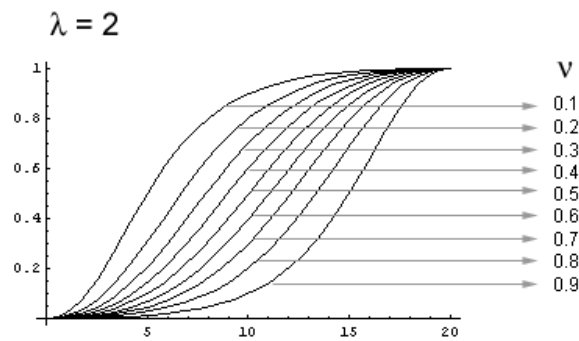


Figure 3.4: S-shaped functions with different inflexion points (ν)

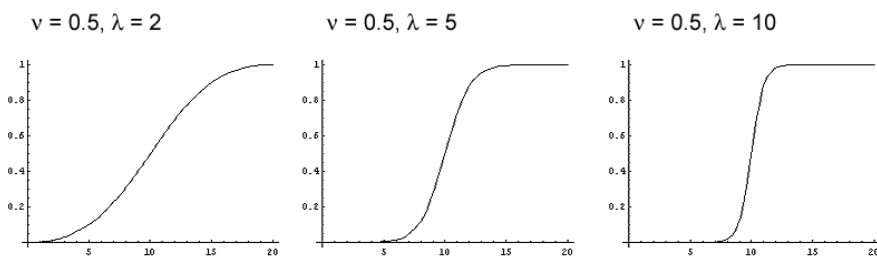


Figure 3.5: S-shaped functions with different slopes (λ)

Membership functions' storage

4.1. Membership function's format

Before storing a membership function, an appropriate storage format has to be found. A membership function is, from a *mEdit* point of view, composed by a list of composing functions. Moreover each composing function is defined by a couple of information:

- its generic function represented by an equation with n parameters ($n \geq 0$) which influence the shape of the composing function. For instance, the generic S-shaped function defines the class of all the S-shaped functions, each of them having a different slope, inflexion point and start/end y-coordinates.
- a set of n values for the generic function's parameters.

A solution to store this pair of information would be to lay it directly in the meta-tables. A structure of tables would fit to the storage of the names of the parameters and the related values, but storing an equation in a table's field is not optimal. This approach would involve a superfluous and time consuming parsing effort each time the *fCQL* interpreter wants to retrieve a membership function. Using a parser also implies defining a grammar to describe the syntax and semantic the equation must respect and this effort is not necessary. Furthermore storing several different S-shaped functions would create redundancy inside the meta-tables.

The storage philosophy of *mEdit* is to separate the generic function from the values of parameters and to save them in different places. On the one hand, *mEdit* proposes to store the generic functions not directly in the meta-tables but inside the editor's code. The generic functions are written inside Java classes which can be instantiated to create new personalized composing functions by setting the parameters' values. On the other hand, the values of the parameters are saved in the meta-tables. The generic equations are fixed and do not need to be edited. Only the values of the parameters vary and that's the reason why it makes sense to store them in the meta-tables.

As the next chapter will explain, the *mEdit* architecture is client-server based. In the *mEdit* solution, the classes representing the generic functions exist inside the server which only needs to retrieve the parameters from the relational database, instantiate a generic function passing the parameters' values and send this object to the client. What the client receives is therefore an instance of a generic function containing the values of the parameters.

To summarize, the idea of *mEdit* is not to store the entire membership function in the meta-tables. The invariant part of the membership function, the generic functions, does not need to be stored in the relational database. But every variable information about the composing functions is saved in the meta-tables: the parameters but also the generic function's type and the definition domain.

4.2. Structure of the meta-tables

In comparison with the structure of the meta-tables proposed for the *fcQL* interpreter in [3], a complete redesigning process has been made. A more logical structure representing the aspects exposed in the previous section was the guideline during the development of the database schema. Figure 4.1 gives an outline of the relations between the meta-tables and figure 4.2 shows in details the columns of the different meta-tables.

The most important table is `linguistic_variable` which lists all the available linguistic variables used by *fcQL*. The link between the linguistic variable and its related attribute is made by the column `attribute_name` and is completed by the foreign key `table_id` determining the table where this attribute is supposed to be declared. The other foreign key `attribute_type_id` defines the type (discrete or categorical) of the considered attribute. Moreover the attributes `domain_start` and `domain_end` respectively determine the lower and higher bounds of the definition domain of the linguistic variable. Finally the attribute `name` is the name of the linguistic variable and normally identifies it. In order to allow an usage of duplicated variable's names related to different tables, the primary key is the `id` attribute.

The table called `tables` indexes all the table's names containing an attribute related to an existing linguistic variable. Since referring to a table using an ID is shorter than with its name, a table is identified by an `id` attribute.

The `attribute_type` table contains actually only two records but might be completed over the time. The *mEdit* editor distinguishes numerical (or continuous) from categorical (or discrete) attributes. The different types of attribute are identified by `id` and also have a `name`.

4.2 Structure of the meta-tables

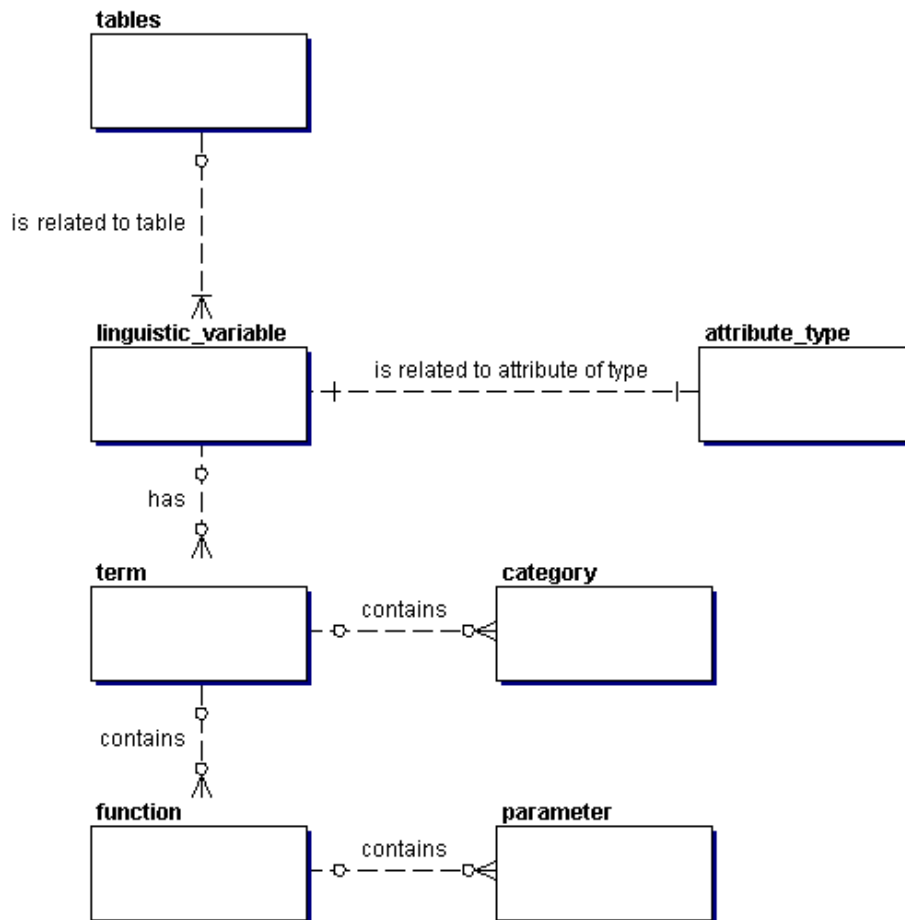


Figure 4.1: The redesigned *fCQL* meta-tables and their relations

The `term` table collects information about all the terms of linguistic variables: the term's name (`name`) and a reference to its linguistic variable (`linguistic_variable_id`). The field `id` is the primary key.

According to the attribute's type, categorical or numerical, each term must have an entry either in the `category` or in the `function` table respectively. The first table stores the name of the categories, their corresponding membership value and a reference to the related term for which the category is defined. Every category has its identifier `id`. The second table, `function`, incorporates all the composing functions forming the membership function which characterizes the referenced term. A composing function is identified by the `id` attribute and can have a descriptive name which is more informative than really relevant in the

4.2 Structure of the meta-tables

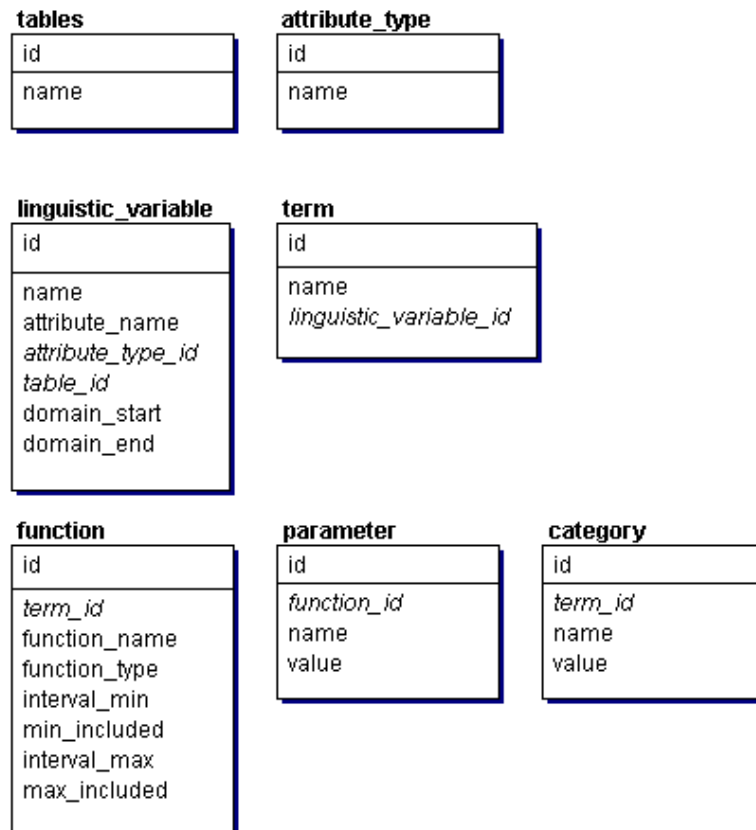


Figure 4.2: Structure of the *fcQL* meta-tables

fcQL architecture. This `function_name` attribute can be used, for instance, to store "gaussian function" for a S-shaped function with a form close to a gaussian. Using words to talk about functions is easier and more convenient than numbers. More important is the type of the function (`function_type`) which indicates the generic function of the composing function. To avoid writing mistakes, the type is not a word but a number. The integer "1" is for the linear function type and "2" for the S-shaped function type. Since a composing function is defined on an interval, the bounds of this interval are given by `interval_min` and `interval_max`. The boolean values `min_included` and `max_included` indicate if the minimum of the interval, respectively the maximum, are included in the interval. The term related to the composing function is referenced by `term_id`. Each composing function with the same `term_id` value defines another part of the same membership function.

The last table, `parameter`, records the names of all the parameters of the

4.2 Structure of the meta-tables

functions contained in the table `function` as well as the corresponding values. It is possible to insert a new function without corresponding parameter or with source code defined parameters. Indeed, for that purpose, no integrity constraint checks that each function has at least one parameter in the `parameter` table. For further details about the structure of the meta-tables, the *SQL* commands creating those tables can be found in appendix B.

Some integrity constraints have been inserted into the relational database schema for the meta-tables. First, parameters without corresponding composing function should not be kept in the database, so a *SQL* integrity constraint deletes every associated `parameters` records when a function is deleted. Then at a higher level, when deleting a term, another integrity constraint removes all related functions or categories according to the attribute's type. Finally, a *SQL* constraint checks that the lower bound of the definition domain of a function is smaller than the higher bound.

mEdit architecture

5.1. Overview

The *mEdit* editor is based on the client-server architecture. Figure 5.1 gives an overview of its structure.

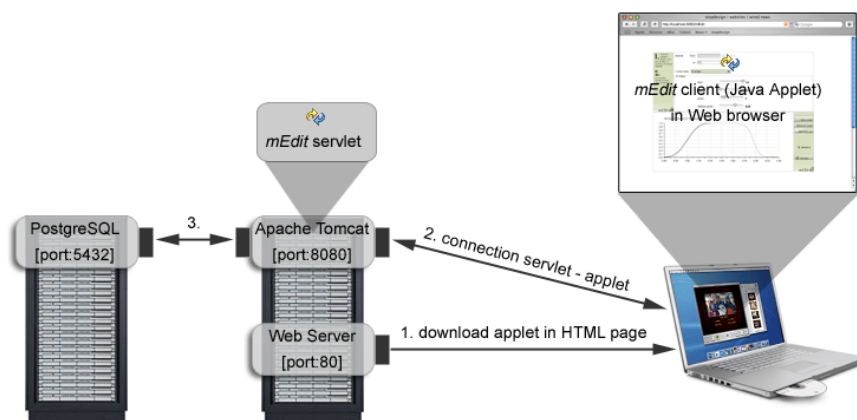


Figure 5.1: *mEdit* architecture

On the user side, the client takes the form of a Java applet embedded in a HTML page which has to be downloaded from a Web server (step 1. in figure 5.1). On the same host as the Web server, a *mEdit* servlet is running on an Apache Tomcat server from the Apache Software Foundation [6] which listens to the port 8080 for *mEdit* client connection requests.

After the client has been downloaded, the applet creates a remote connection (step 2. in figure 5.1) to the servlet which should be located on the same host as the Web server for security matters. Indeed, if the servlet is not running on the host that provides the user's browser the HTML page and the applet, this latter is not allowed to open a connection to the servlet. To permit this operation, the

5.2 Applet-Servlet communication

Java Security Manager should be configured in a way that it reads the permission granted to the applet to connect to a third host. On Java platforms that are compatible with JDK 1.2, the permission must be granted by an entry in a policy file. More information about security aspects can be found in [4].

Then, every operation on the relational database that the applet requests passes through the servlet which forwards the request to the PostgreSQL server (step 3. in figure 5.1). No direct communication between the applet and the database server is allowed. The unique access point to the database is the servlet.

5.2. Applet-Servlet communication

To guarantee that the connection between the applet and the servlet can be established, *mEdit* does not use sockets. Often firewalls do not allow socket connections whereas HTTP connections are permitted. To highlight a recent technology, the servlet could have been implemented using Remote Method Invocation (RMI), but most of the Web browsers do not support this technology or need an extension to work properly. So in view of the fact that the traffic between the servlet and the applet is neither heavy nor frequent, a simple and widely supported HTTP communication channel is an appropriate choice.

The communication follows an ad hoc protocol which allows the client to ask the servlet for one of five actions by sending one or several parameters through a GET HTTP request. The standard URL of such a request is the following:

```
http://host/mEdit/start?action=actionName&lingVar=linguisticVariable&term=term
```

The URL to access the servlet is defined in the `web.xml` file (see subsection 7.2.2). By replacing the *actionName* by one of the following strings, the applet can ask the servlet for an action:

- `connection_test`: the connection applet-servlet is tested. If the test request sent by the client arrives successfully, the servlet will also test its connection to the relational database. If a problem occurs or if the message does not reach the servlet, the test fails and the applet is informed by receiving a boolean status. This status is "true" if the test succeeds and "false" otherwise.
- `get_all_terms`: reacting to this message, the servlet sends a serialized `Hashtable` containing all the linguistic variables stored in the meta-tables and their corresponding vector of terms. The first element of this vector is actually the type of the related attribute. If no term is found, an empty

Hashtable is sent back to the client. Neither *linguisticVariable* nor *term* is required in the request because this action does not need this information.

- `get_mFunction`: by sending this message together with a specific linguistic variable and term, the client wants to get the corresponding membership function. The answer formulated by the servlet is actually a serialized `MFunction` object. More information about this class follows in the next chapter depicting the *mEdit* Java library. If no membership function is defined for the specified linguistic variable and term in the meta-tables, the servlet returns an empty `MFunction` object, that is containing only the definition domain but no composing function.
- `delete_term`: after reception of this message with specific linguistic variable and term, the servlet deletes all the information related to this term, including its entry in the `term` table. If the term is related to a categorical attribute, all its categories in the meta-tables are erased. Otherwise, if the term is related to a numerical attribute, all its functions and their corresponding parameters are deleted. This is the effect of the integrity constraints of the meta-tables. The client finally receives the status of the operation as a boolean "true" if the action was successful or "false" otherwise.
- `get_categories`: the servlet returns a `Hashtable` containing all the categories names and their corresponding membership value for the specified linguistic variable and term. If no category is found in the meta-tables, an empty `Hashtable` is sent back to the client.

A sixth action may be requested by the client in a POST request: when the client needs to store a defined membership function or a set of categories, it can send it as a serialized Java object to the servlet which stores it in the meta-tables. If a set of categories has to be stored, a serialized `Vector` is sent: the first element is the term, the second the linguistic variable and the third a `Hashtable` of categories and their membership value. If the object is a membership function, a serialized `MFunction` is sent. As the next chapter will show, this object contains also the linguistic variable and the term that the membership function describes. Once the definition is stored, the servlet sends the client a boolean value to indicate that the operation succeeded or failed.

5.3. Servlet-Database communication

When the servlet starts, it tests the connection with the relational database whose name, location, port and access information are specified in the `database.properties` file located in the `/webapps/mEdit` directory inside the Tomcat server. For each operation requested by the client on the database, the servlet opens a connection to the database with the password and

5.4 *mEdit* activity diagram

username specified in the `database.properties` file. Every operation on the database is done by a part of the servlet called the *mEdit* database controller. This Java class operates the six above actions that the client can request.

The *mEdit* servlet utilizes a JDBC driver of type 4 using the JDBC API version 3.0 and provided by the PostgreSQL Development Group for PostgreSQL 7.3. No typical `DriverManager` is used, but a J2SE 1.4 conformed `DataSource`, the class `org.postgresql.jdbc3.Jdbc3SimpleDataSource`.

5.4. *mEdit* activity diagram

For a better comprehension of the different steps of *mEdit* when using the client, figure 5.2 graphically describes the different activities of the editor. To show the interactions between the server and the client, the diagram is split in two parts: the activities on the client side and those on the server side. Most of the activities occur on the client side, but the server also has some key activities. Therefore let us describe step by step the path the user follows through the activity diagram.

- *Start Client*: When launching the client, the connection to the server is first initialized and the server starts the *Retrieve Terms and Linguistic Variables* activity on the client's demand.
- *Retrieve Linguistic Variables and Terms*: the server retrieves the names of the terms and linguistic variables from the meta-tables and sends this information back to the client. This activity corresponds to the `get_all_terms` servlet action introduced in the previous section.
- *Choose Linguistic Variable and Term*: the client displays the names of the terms and linguistic variables received from the server in the graphical user interface. The user has to choose a linguistic variable and a term to work on. Before going to the next activity, the user has to decide which operation he wants to perform on the term:
 1. edit / create the term
 2. delete the term

If he chooses to edit or create a term, the server will start the *Retrieve Definition* activity. Otherwise the server proceeds with the *Delete Term* activity.

- *Delete Term*: if the user wants to delete an existing term, the request is sent to the server which performs the `delete_term` servlet action. After having deleted a term, the server launches the *Confirmation* activity on the client.

5.4 *mEdit* activity diagram



Figure 5.2: *mEdit* activity diagram

5.4 *mEdit* activity diagram

- *Retrieve Definition*: if the user chooses to edit an existing term or to create a new one, the client asks the server for a `get_mFunction` or `get_categories` servlet action depending on the type of the associated attribute.
- *Define Membership function*: this activity uses most of the *mEdit* resources. When the client receives the existing membership function or the structure to store a new one from the server, the user can define the membership function with the help of the graphical interface of the *mEdit* client.
- *Define Categories*: this step is planned to edit the categories and the membership values the client receives from the server or to define a new term related to a categorical attribute. As for the definition of a membership function, the graphical user interface allows the user to edit the categories and their membership value in a simple way.
- *Store Definition*: once the server gets the definition to store in the meta-tables from the client, the `store` servlet action is launched on the server.
- *Get Confirmation*: after having stored or deleted a definition, the server sends a confirmation to the client. The message is displayed in the client interface. Depending on the success of the storage/deletion operation, the user has two possibilities:
 1. if the storage/deletion operation succeeded or if the deletion operation failed, the user can either quit the client or continue the edition of other terms. By choosing to continue, he starts again the *Settings* activity. Otherwise, the user goes to the *Leave Client* activity.
 2. if the storage operation failed, the user can either quit *mEdit* or go back to the previous activity of the client. More details will be given in chapter 7.
- *Leave Client*: finally, this last step ends the flow of activities and closes the connection between the client and the server. No coming back is possible without reloading the client.

Communication problems may occur between the client and the server. However, the *mEdit* is implemented to manage such situations which might cause the client to crash. If the server does not respond or if the connection fails, the flow of activities is redirected and the user is informed by a message in the client interface. If trouble occurs before the definition of a membership function or categories, the client goes directly to the *Leave Client* activity. Otherwise, communication problems are considered as a storage/deletion failure.

mEdit Java library

The complete Java library of *mEdit* is provided in a Java archive package named `mEdit.jar`. This compressed file, as figure 6.1 shows, is composed by three different packages: `mEdit`, `com.incor` and `edu.rit`.



Figure 6.1: The `mEdit.jar` file composed by three packages

6.1. The *mEdit* package

This package includes almost all the Java classes of *mEdit*. Its structure was thought for future extensions. Therefore the classes are distributed among several subpackages which reflect the architecture of the editor. Figure 6.2 provides a caption to understand the following UML class diagrams.



Figure 6.2: UML caption for the class diagrams

6.1.1. `mEdit`

This main package contains the most important classes and subpackages which gathers the *mEdit* main logical entities like the client, the servlet, the membership functions and so on. Figure 6.3 shows its class diagram.

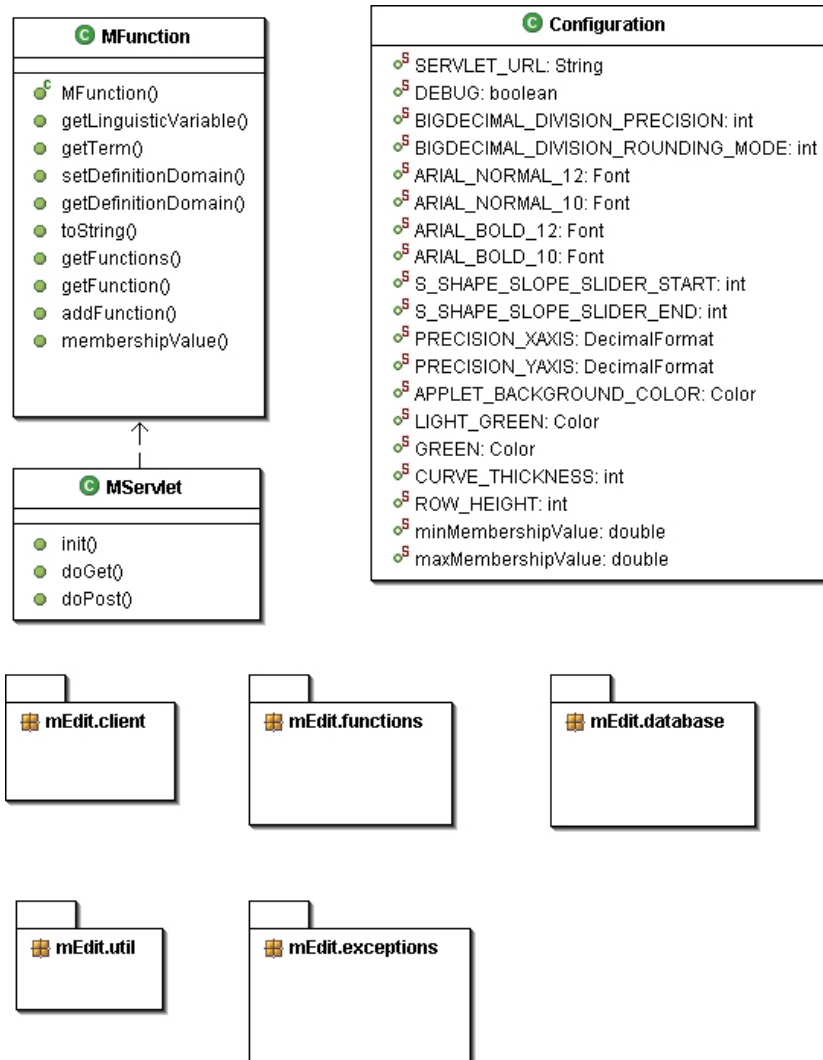


Figure 6.3: The *mEdit* package

In the *Configuration* class, there are some general parameters that can be set by the developer and which influence the whole editor. Those parameters imply changes on:

- the client applet’s layout: fonts, colors, graduations of axis, graduations of sliders and so on.
- the debugging: the debugging output can be enabled or disabled.
- the URL of the servlet so that the client can connect to it.

- the division of `BigDecimal`: since *mEdit* computes with real numbers and uses the `BigDecimal` class of the `java.math` package, the precision when dividing is very important to ensure accuracy. Different modes of rounding can be set.

Another very important class, `MFunction`, represents the structure of a membership function. It contains methods to access its composing functions (`getFunction`, `getFunctions`) and definition domain (`getDefinitionDomain`, `setDefinitionDomain`), to get the name of the related term and linguistic variable and to compute the membership value.

The *mEdit* server side is represented by the `MServlet` class which implements all the methods required in a servlet. All the client's classes are placed in the `mEdit.client` package. The database controller is in the `mEdit.database` package and all the classes related to the functions are in the `mEdit.functions` package. Some useful tools are located in `mEdit.util` and the small `mEdit.exceptions` package aims to contain all the *mEdit* specific exceptions.

6.1.2. `mEdit.client`

This subpackage provides all the graphical components for the client/applet as figure 6.4 shows.

The `StandardPanel` offers a template for other panels. All the subclasses of `StandardPanel` represent a separated activity of the `mEdit` client. The `SettingsPanel` is the window which allows the user to choose the term to work on and the operation (deletion, creation or edition). The `CategoryPanel` is for the definition of categories and the `ConfirmationPanel` displays the confirmation message after a storage or deletion operation. Then the `FinalPanel` represents the *Leave Client* activity and the `ErrorPanel` informs the user if a communication problem happens before a definition activity. A particular panel, `DefinitionPanel`, has an extended environment to allow the graphical definition of membership functions.

The `SortButtonRenderer`, `MyTableModel` and `TableSorter` classes are components of the `CategoriesPanel` window. The `StandardMiniPanel` and the `StandardButton` are reusable components used in different panels.

Extending the `JApplet` class, the main class of the client is `MClient`. Besides the required applet's methods, the remaining method controls the different activities of the client. For instance, the `onceAgain()` method restarts the

6.1 The *mEdit* package

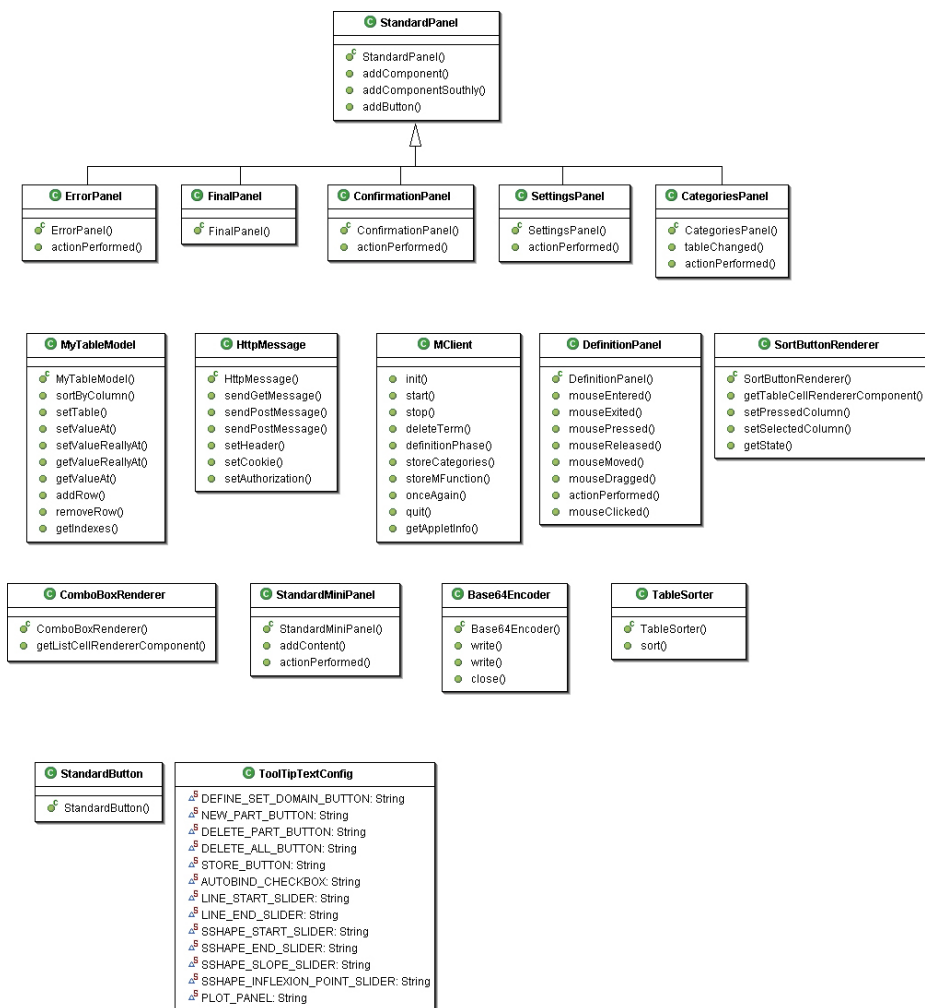


Figure 6.4: The `mEdit.client` Java package

Choose *Linguistic Variable and Term* activity after reception of a successful confirmation and the `quit()` method launches the *Leave Client* activity.

The HTTP communication with the servlet is managed by the `HttpMessage` class which employs the `Base64Encoder` class to properly encode the message in the request. Finally, the `ToolTipTextConfig` class configures the important tool tip texts of the client's interface.

6.1.3. `mEdit.database`

This subpackage contains only the database controller whose methods execute the six actions of the servlet presented in section 5.2. Actually two methods are available to store a term depending on its attribute's type: `storeCategories` for categorical attributes and `storeMFunction` for continuous attributes. The `isConnectionToDatabaseOK` method does not only test if the servlet communicates properly with the database but also checks if the applet is connected to the servlet. The `DBController` class is instantiated only in the `MServlet` class when the servlet starts.

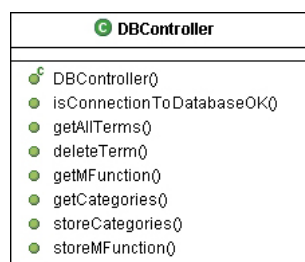


Figure 6.5: The `mEdit.database` Java package

6.1.4. `mEdit.exceptions`

A short `IntervalException` class is present in this subpackage illustrated by figure 6.6. This exception is thrown in the definition panel if the user enters a incorrect maximal bound for the definition domain of a composing function. The maximal bound of a composing function must be greater than its minimum, smaller or equal to the minimum of the next composing function and smaller or equal to the definition domain's maximum of the membership function.

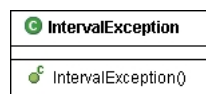
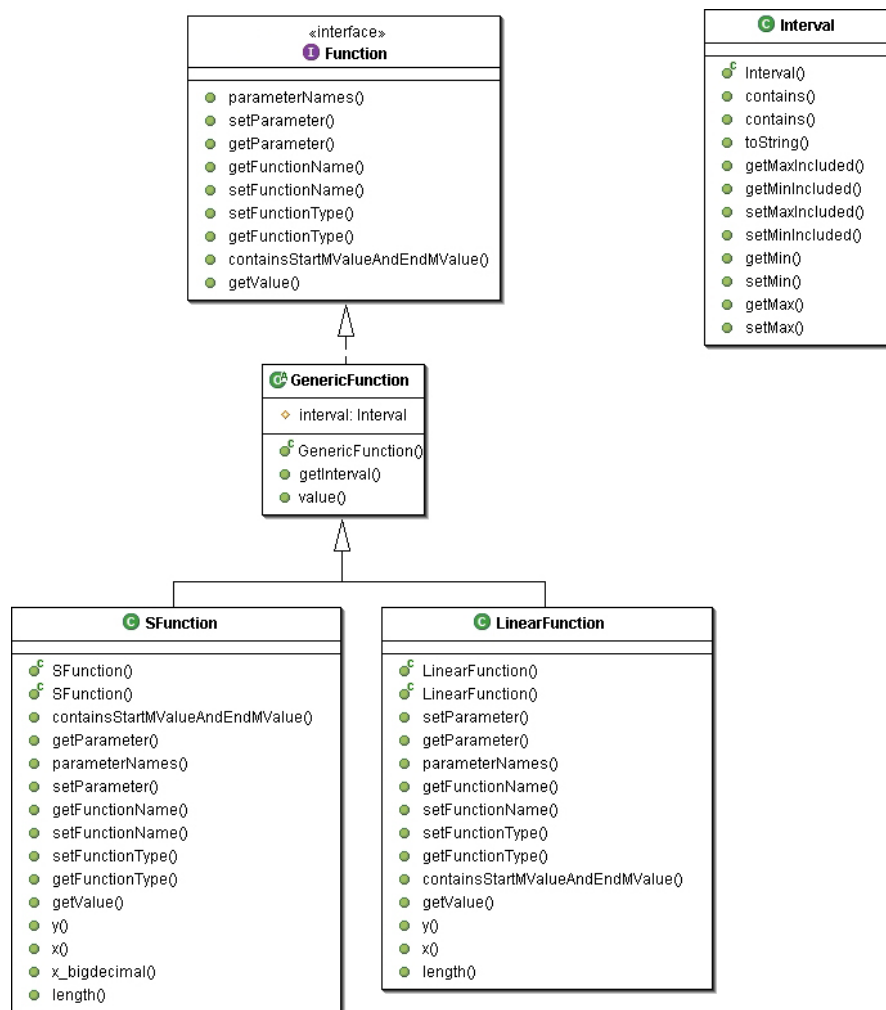


Figure 6.6: The `mEdit.exceptions` Java package

6.1.5. `mEdit.functions`

As figure 6.7 shows, this subpackage gathers the classes related to the membership functions and the generic functions.

Figure 6.7: The `mEdit.functions` Java package

The `GenericFunction` abstract class consists in a frame for all generic and therefore all composing functions of a membership function. It implements the `Function` interface which incorporates the accessors for the name, type and parameters of the composing functions. To preserve consistency, the names of the parameters utilized by this interface must be the same as the ones of the `parameter` meta-table. The editor uses indeed this interface to retrieve the name of the parameters and then to write them in the meta-table.

Extending the `GenericFunction` class, the `LinearFunction` and `SFunction` classes code the equation of the two generic functions defining the shapes that the *mEdit* user can draw (see section 3.3). Not displayed in figure 6.7,

6.2 The *edu.rit* package

the `GenericFunction` class also implements the `XYSeries` interface of the `edu.rit.numeric` package. More details about this helpful package follow in the next section.

Finally, the `Interval` class provides the appropriate structure to define and edit a definition domain for a composing function or a membership function.

6.1.6. `mEdit.util`

This last subpackage offers some tools represented in figure 6.8. For instance, a particular Layout Manager (`VerticalFlowLayout`) or a scale converter (`ScaleConverter`), which converts a real number between 0 and 1 into a integer between 0 and 100 (x100) and the opposite are used in the *mEdit* client. Last but not least, the `Debug` class is the output channel for debugging messages.

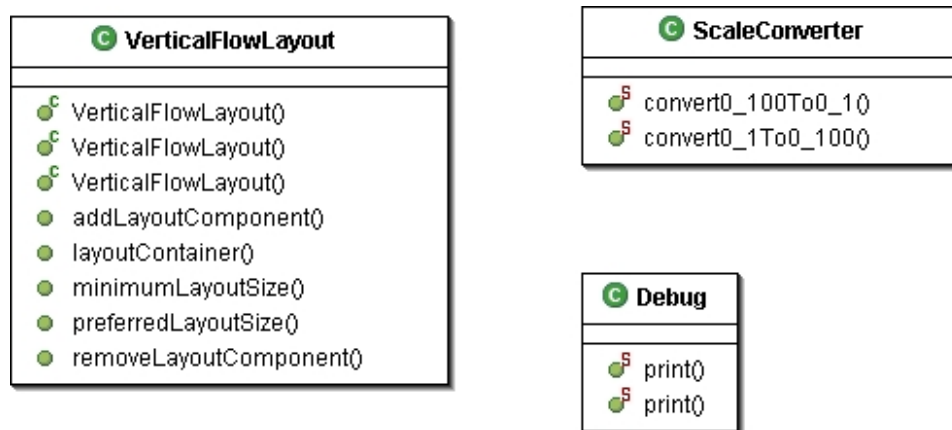


Figure 6.8: The `mEdit.util` Java package

6.2. The *edu.rit* package

This package is a course library from the Computer Science Department at the Rochester Institute of Technology and can be used under the terms of the GNU General Public License [7]. More information about the developers and the library is available online¹.

¹ <http://www.cs.rit.edu/~ark/cscl.shtml>, last visited on January 28, 2004

6.2 The *edu.rit* package

Actually the *mEdit* editor only exploits a tiny part of the original `edu.rit` package. The used `edu.rit.numeric` package contains helpful classes and interfaces for scientific numerical computation and its subpackage `edu.rit.numeric.plot` provides tools for plotting series of points. Figures 6.9 and 6.10 give a short overview of its content.

The most relevant file is located in `edu.rit.numeric.plot`. The `XYSeries` interface extending `Series` represents a serie of points that can be painted on a plot using the classes of the `edu.rit.numeric.plot` package. A drawable *mEdit* function is therefore made of a serie of points. That's why the `mEdit.functions.Function` class implements `XYSeries`.

To draw a serie of points on a plot, a class implementing the `XYSeries` interface must be passed to a `XYPlotSeries` object. This latter will be added to a `XYPlot` object using its `addPlotSeries` method. Different graduated axis can be chosen among `NumericalAxis`, `LinearAxis` and `LogarithmicAxis`. Ticks can be drawn on the axis, the exact thickness of a curve can be set and labels can be pasted on the plot. Thus these classes and interfaces offer a large range of graphical settings and generate really good-looking graphs.

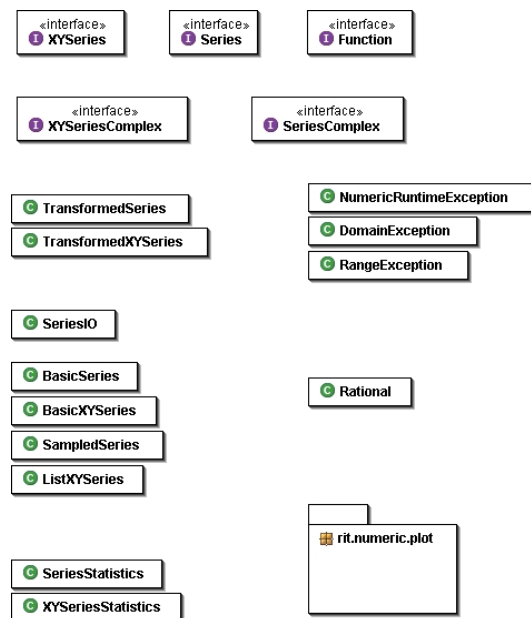


Figure 6.9: The `edu.rit.numeric` Java package

6.3 The *com.incors* package

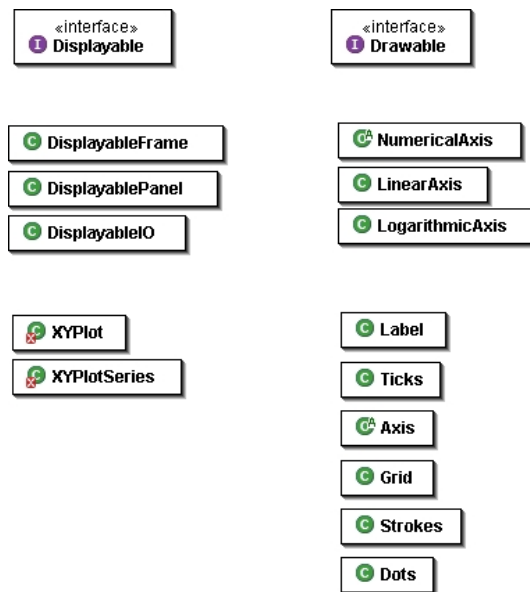


Figure 6.10: The `edu.rit.numeric.plot` Java package

6.3. The *com.incors* package

The Java files of the *com.incors* package used in *mEdit* are precisely in the `com.incors.plat.kunststoff` directory. Those classes contribute to the Java Look&Feel of *mEdit* and were written by INCORS GmbH². This Look&Feel, called *Kunststoff*, is an extension of the Java Metal Look&Feel and this open source project is protected by the GNU Lesser General Public Licence [8]. The *Kunststoff* package is used by the `MClient` class, but the client overwrites some properties of this Look&Feel and manages its own colors in the `mEdit.Configuration` file.

² <http://www.incors.com/>, last visited on January 28, 2004

Using *mEdit*

7.1. Software requirements

7.1.1. Java Runtime Environment

The *mEdit* editor is an application developed in Java. Therefore to run it, the user needs the Java Runtime Environment (JRE). First of all, to make certain that the communication between the applet and the servlet works properly, the Java Runtime Environment has to support the serialization of Java objects. To guarantee this feature, the JRE must be at least a 1.1 version.

A more restrictive requirement comes from the PostgreSQL driver for PostgreSQL 7.3 used by the database controller and running on the servlet. The chosen version of this driver, `org.postgresql.jdbc3.Jdbc3SimpleDataSource`, is conformed to the JDBC 3.0 specification. According to the official PostgreSQL documentation for developers, the JDBC 3.0 driver is intended for JDK 1.4 environments and will not run under JDK 1.1, 1.2 or 1.3 [13]. Therefore the minimal required version of the JRE is the version 1.4. By changing the PostgreSQL database driver¹, the servlet might run with an older JRE. Actually, the *mEdit* editor was developed with J2SE 1.4.2 beta 2.

7.1.2. Servlets container

To run the *mEdit* server, a servlet container is required. The application was tested with one of the latest Apache Jakarta/Tomcat server release, the 5.0.12 version, which supports the 2.4 servlet specification. Because the *mEdit* servlet does actually not use this new specification, it runs also correctly on an older version of Apache Tomcat. Other open source servlets containers or web servers supporting servlets could be used:

- Resin by Caucho²

¹ More JDBC drivers for PostgreSQL under <http://jdbc.postgresql.org>, last visited on January 28, 2004

² see <http://www.caucho.com>, last visited on January 28, 2004

7.2 Installation

- Jigsaw Server by the World Wide Web Consortium³

Besides the open source solutions, other profitable web servers are available. In addition, most of the applications servers also run servlets environments.

The Apache Tomcat server might also be utilized as a web server. It would give a remote access to the HTML page containing the applet. In this case, no web server is necessary.

7.1.3. Web server

If the applet runs on the same host as the servlets container, there is no need of web server. Otherwise, two solutions could be considered:

- either the servlets server operates also as a web server and no web server is required,
- or a web server is running on the same host as the servlets container.

Therefore a conventional web server is not really necessary, but often offers more features, like security features, than a servlets server acting also as a web server.

7.2. Installation

The *mEdit* installation CD-ROM is available in appendix C. As figure 7.1 shows, the *mEdit* editor's installation package includes three directories: *Applet*, *Servlet* and *PostgreSQL*.

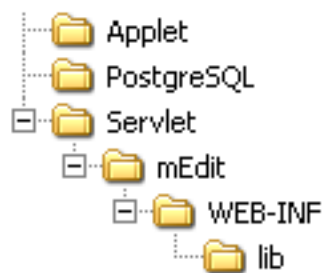


Figure 7.1: The *mEdit* installation package

³ see <http://www.w3c.org/Jigsaw>, last visited on January 28, 2004

7.2.1. Installing the meta-tables

The *mEdit* installation files in the *PostgreSQL* directory provide some useful database scripts. The first two scripts, *startdb.sql* and *stopdb.sql*, can be used to start and stop the PostgreSQL server. They have to be first configured to define the right database's path on the hard disk and the server access properties (hostname and port).

The most relevant are the *install.sql* and *desinstall.sql* scripts which must be called inside the console using the following command line:

```
psql database_name > script_name
```

where `psql` is a command representing the terminal-based front-end to PostgreSQL. The installation script *install.sql* creates the meta-tables introduced in section 4.2 and fills them with some example values. The second script removes those tables from the specified database. Before starting the servlet, the PostgreSQL server must be running and the meta-tables must be installed.

7.2.2. Installing the servlet

The servlet files are located in the *Servlet* directory (see figure 7.1). A subdirectory called *mEdit* contains all the configuration files and the Java classes. If the servlet runs on an Apache Tomcat Server, the whole *mEdit* directory must be put into the *webapps* directory of Tomcat.

The *mEdit* directory includes the *WEB-INF* directory and a configuration file named *database.properties*. This latter file indicates where the servlet can find the database (host, port and database's name) and what are the username and password to access it. This file is parsed when the servlet starts. Figure 7.2 shows an example of this file and the syntax it should respect.

In the *WEB-INF* directory, the standard configuration file *web.xml* stores the library deployment descriptor. No change has to be made in it, especially concerning the servlet main class and the URL mapping to communicate with the servlet. The required libraries are located inside the *lib* directory. The first jar file is the *mEdit.jar* library presented in the previous chapter. The second one, *pg73jdbc3.jar*, contains the JDBC 3.0 version of the PostgreSQL 7.3 driver.

7.2.3. Installing the applet

The two files needed to start the client are located in the *Applet* directory. The first file is a HTML page named *applet.htm* which includes the call to the

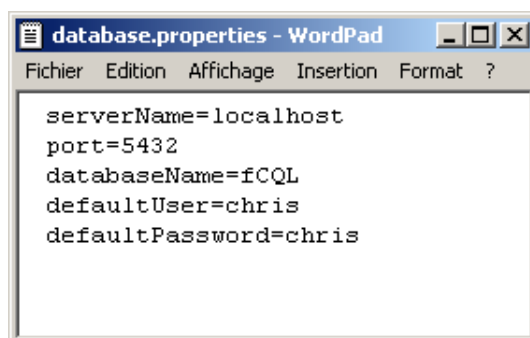


Figure 7.2: The *database.properties* configuration file

`mEdit.client.MClient` applet. All the applet's classes are stored in the second file, the essential `mEdit.jar` library. Before starting the client, the servlet should be listening.

The applet is 800 pixels wide and 600 pixels high. For the sake of comfort, it would be agreeable that the screen resolution ensures a display of the applet without scrolling.

7.3. Tutorial

7.3.1. Choosing a linguistic variable and a term to work on

After having started the PostgreSQL server and the servlet, the client can be launched to use the *mEdit* editor. The first window which appears (see figure 7.3) represents the *Choose Linguistic Variable and Term* activity and the decision on the type of the operation to perform. The user is invited to choose a linguistic variable in the scrolling list (1). All the existing linguistic variables in the meta-tables are displayed.

After having chosen a linguistic variable, the user has to decide which related term he wants to work on by clicking in the second scrolling list (see figure 7.4). If the user changes the linguistic variable, this second list is updated and displays the terms related to the newly chosen variable. The user has the choice between the two following actions:

- define a new term by writing its name in the text field (2). If the new term already exists in the meta-tables, a warning message appears in this text field when the user clicks on the "next" button.
- edit an existing term in the scrolling list (3). An editing option is to delete the

7.3 Tutorial

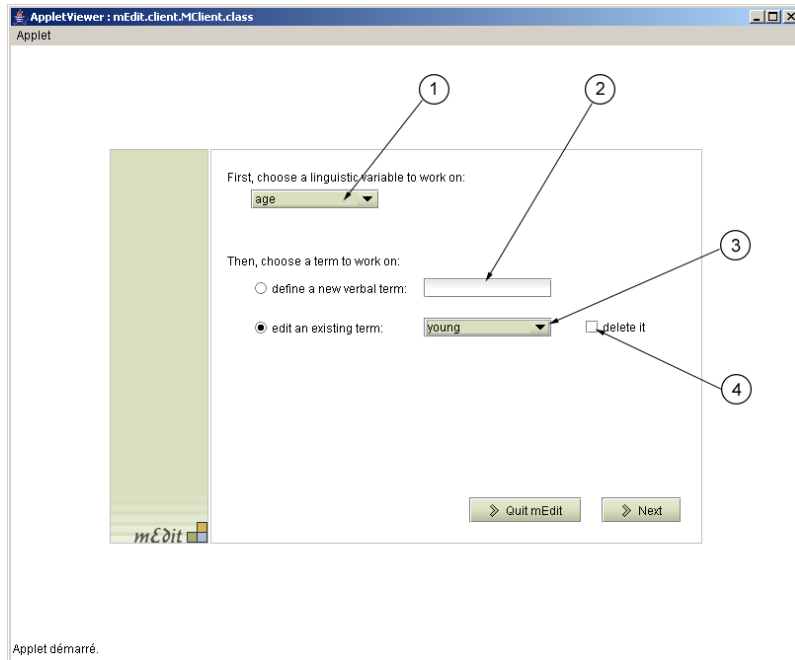


Figure 7.3: The settings window - first window appearing when starting the applet



Figure 7.4: The two scrolling lists for the linguistic variables and their related terms

term by clicking in the "delete it" case (4). The user must be aware that the term will be definitely erased from the meta-tables if he chooses this option and clicks on the "next" button. No confirmation is asked.

If the user clicks on the "quit" button, the client ends with the *Leave Client* activity represented by a final window displaying the author's name and his email address. On the other hand, clicking on the "next" button leads to the next activity and window.

7.3.2. Defining categories

If the attribute related to the chosen linguistic variable is categorical, a window like shown in figure 7.5 appears after having clicked "next" in the settings window.

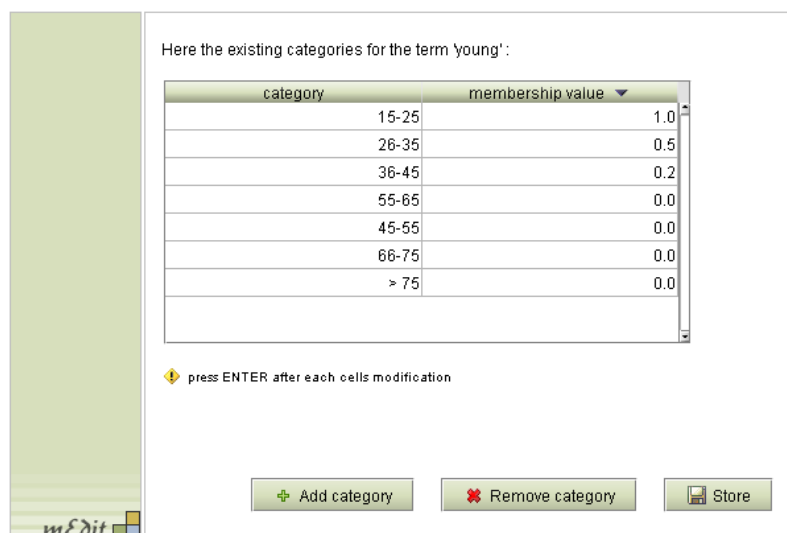


Figure 7.5: Defining the categories and their membership value

All the available categories for the chosen term are listed in the left column of the table. Their corresponding membership values are displayed in the right column and are sorted in descending order. A click on a cell enables the edition of a category or a membership value. Do not forget to press the enter key after having edited a cell, otherwise the change will not be stored. By pressing on the arrow beside the header label of the right column, the membership values are sorted in ascending order. To remove a category, the user has to click on a cell, then press the "remove category" button and the whole row will disappear. To add a new category, he has to click on the corresponding button and fulfil the new row with a

category name and its membership value.

Control mechanisms prevent some classical and recurrent mistakes. For instance, if the user enters a membership value that is not between 0 and 1 or a non-numerical value, the editor replaces this value by a 0.0 value. Also if no value is entered, a 0.0 value is automatically set. For categories, if the user enters a duplicated category name, the editor displays in the cell a "#DUPLICATED VALUE#" error message.

The defined categories and membership values are finally stored in the relational database by clicking on the "store" button. Categories with empty name, "#DUPLICATED VALUE#" name or no membership value are not stored.

7.3.3. Defining a membership function

If the attribute related to the chosen linguistic variable is numerical, a window as shown in figure 7.6 appears after having clicked "next" in the settings window.

The instructions on the left side of the window are clear: first the user has to click on a function composing the membership function displayed at the bottom of the window. When the user moves the mouse cursor on the plot, the cursor changes into a hand. If he clicks on a function, the selected function changes into a dark green dotted line. It means this function is selected. Then he might change the interval of the selected function and finally its type. Two general situations can append:

- the user chooses to create a new membership function. In this case, the plot is empty and no function is displayed. The user has to click on the "new part" button and a linear function will appear on the whole definition domain. Then only he can change the interval and the type of this function.
- the user chooses to edit an existing membership function. Then the user can click on a part of the membership function and change its interval and type.

The different interactive components of the definition window are the following:

- **interval text field "from"**: this field cannot be edited, but shows the definition domain's start for the selected function.
- **interval text field "to"**: when a function is selected, this field shows the end of its definition domain. Unlike the "from" field, the value can be edited. If a new value is inserted, the user has to type the enter key to update the plot. The start x-position of the next function is automatically fitted to avoid horizontal gaps. A control mechanism ensures that the user does not enter a

7.3 Tutorial

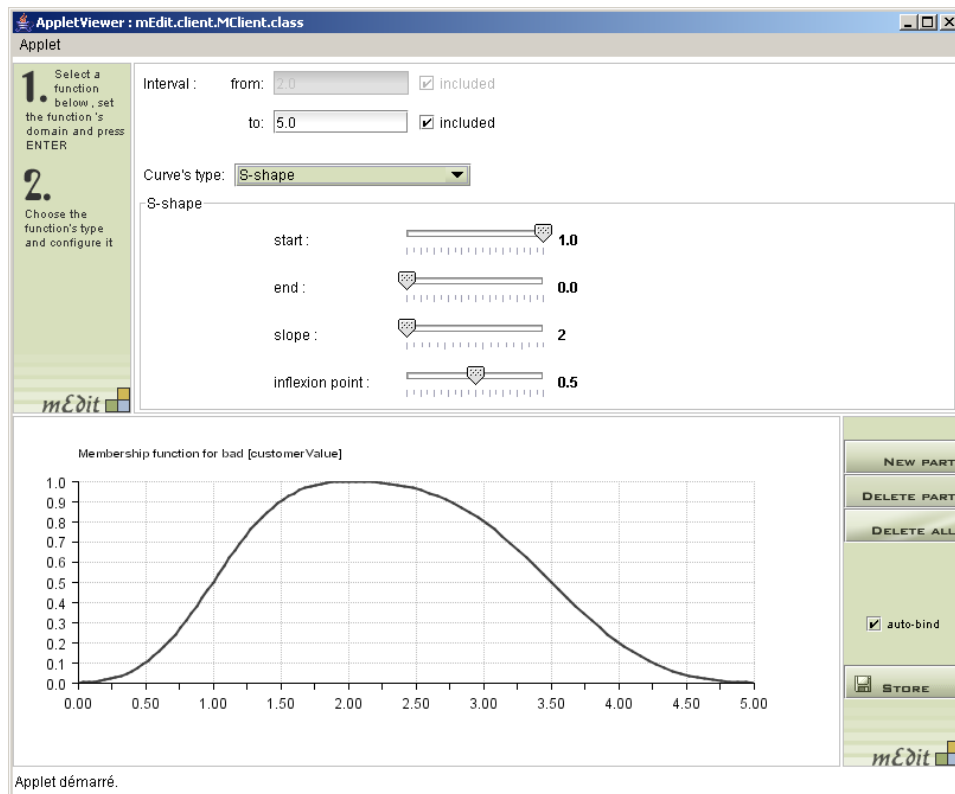


Figure 7.6: Defining a membership function

value which is not in the definition domain of the membership function. If a wrong value is typed, an error message is displayed in this text field and the value is not taken into account.

- **”included” checkbox related to ”from”**: this checkbox cannot be edited. If a function is selected, it indicates if its definition domain’s start is included.
- **”included” checkbox related to ”to”**: if this checkbox and a composing function are selected, it means that its definition domain’s end is included.
- **curve’s type scrolling list**: if a composing function is selected, its type is selected in the list and the user can change it, but the values of the ”start” and ”end” sliders keep the same.
- **sliders**: using those sliders, the user sets the parameters of the selected function. The changes occur in real time on the plot and the value of the parameters is displayed on the right side of the sliders.
- **”new part” button**: this button creates a new composing function from the end of the last defined function up to the end to the definition domain of the

membership function. This way of doing avoids horizontal gaps inside the membership function which must be defined on its entire definition domain. If the membership function is already completely defined, clicking on this button does not affect it. The standard type of a new part is linear.

- **”delete part” button:** this button deletes the selected composing function. If no function is selected, nothing appends. Since the membership function must be defined on the entire attribute’s domain, the interval lower bound of the composing function following the deleted function is automatically set to the interval lower bound of the deleted function.
- **”delete all” button:** this button clears completely the plot. No composing function remains.
- **”auto-bind” checkbox:** when the checkbox is selected, the editor keeps the continuity of the membership function. If the user deletes a composing function or changes its y-value, the y-value of the neighboring function is automatically adjusted so that no jump occurs.
- **”store” button:** if the membership function is defined on the entire definition domain, the editor stores its definition and goes to the confirmation window. Otherwise a small error window is displayed as shown in figure 7.7.

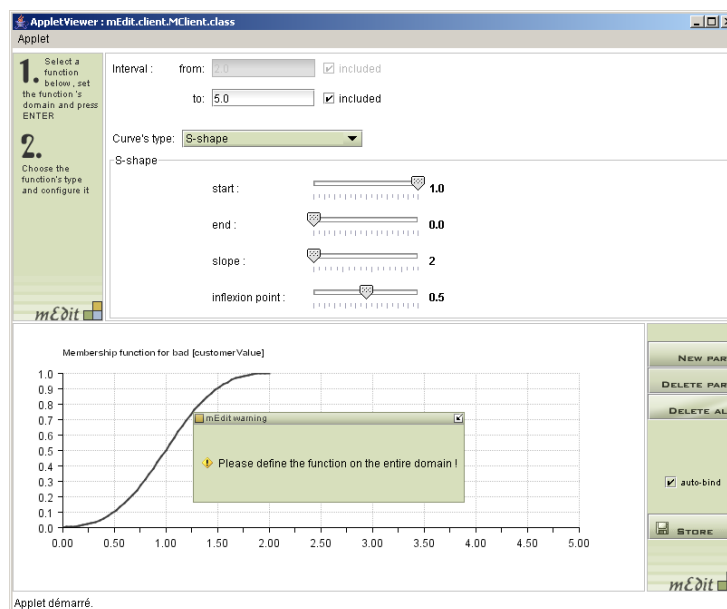


Figure 7.7: Error message when a membership function is not fully defined

7.3 Tutorial

To improve the simplicity of use of *mEdit*, some helpful tool tip texts are placed inside the definition window. Figure 7.8 shows the some examples of those tips.

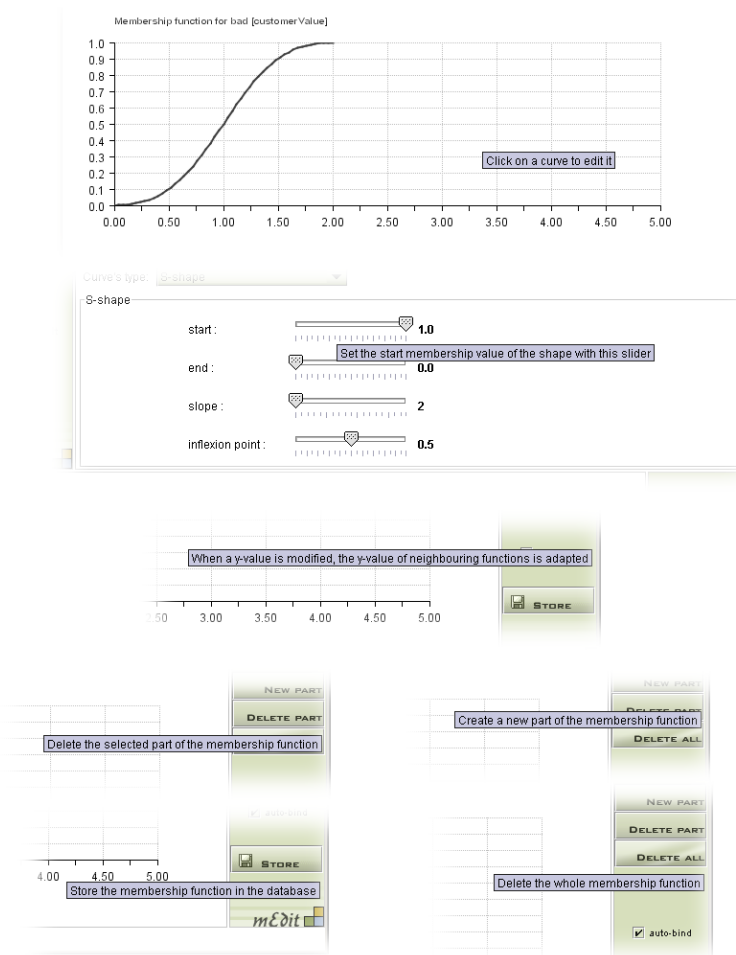


Figure 7.8: The tool tip texts for a simple use of *mEdit*

7.3.4. Getting the confirmation

A confirmation window appears in three situations:

- after having pressed the "store" button in the categories window,
- after having pressed the "store" button in the definition window,
- after having chosen to delete a term in the settings window and having pressed the "next" button.

A confirmation can be positive or negative. A positive confirmation indicates that the chosen action (either a store or a delete operation) has succeeded. The confirmation is negative either if the communication with the servlet, respectively the database, was interrupted or if the RDBMS crashes during the treatment of the query. Figures 7.9 and 7.10 show all the possible windows for positive, respectively negative confirmations.



Figure 7.9: The two kinds of positive confirmations



Figure 7.10: The two kinds of negative confirmations

The advantage of the negative confirmation after having defined or edited a term is that the user can press the "go back" button. This will display the previous window (the definition or categories window) with the previously defined membership function or categories. If meanwhile the problem has been resolved, the user can try once again to store the definition. So thanks to the "go back" button of this confirmation window, no information is definitely lost. For deletion operations, if the term could not be deleted, no "go back" button is displayed because no information is lost. The user has to try again.

By clicking on the "continue" button, the settings window will be displayed and the user can edit another term. By clicking on the "quit mEdit" button, the editor will display its last window: the final window.

7.3.5. Quitting *mEdit*

When the user chooses to leave *mEdit* a last window is displayed: the final window (see figure 7.11). It contains the full name of the author, the department and the university where the editor was developed and an email address for contact.

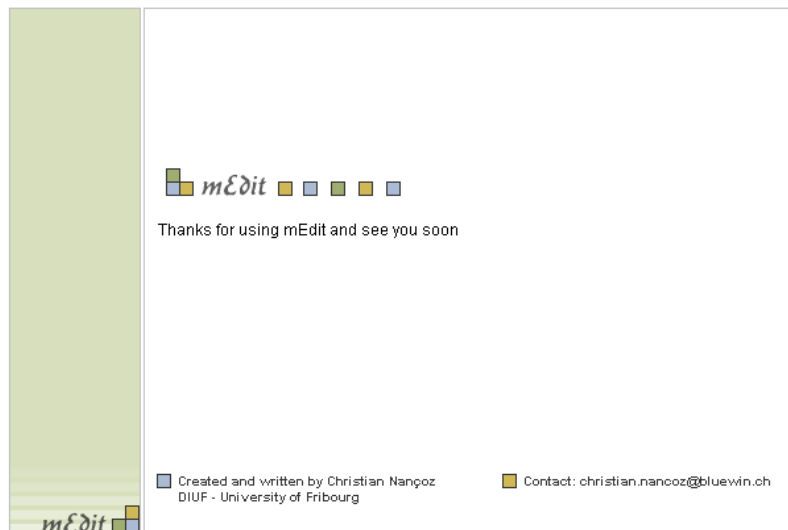


Figure 7.11: The final window

Once this window is shown, no button allows to go back to the settings window. To restart the editor, the user has to either reload the HTML page or restart the applet.

7.4. For developers

7.4.1. Source code documentation

The *mEdit* editor is provided with a complete Bravado documentation. All methods, constructors and fields are described whatever visibility they have. Moreover the source code is accurately commented so that developers who might extend

the editor can quickly make changes to the code. Therefore the Bravado, the well commented source code and the logical structure of the *mEdit* library allow an easy comprehension of the Java classes and their interaction.

7.4.2. Adding a new curve's type in *mEdit*

A curve's type is actually referring to a generic function. The notion of "curve's type" is used for the user in the *mEdit* interface, but means "generic function". A big issue of this work was the way to define the membership functions. The short but smart panel of generic functions (linear and S-shaped) of *mEdit* allows the definition of a wide range of membership functions. It may be helpful to have access to another type of curves. For instance, if the business user notices that a gaussian curve is necessary, the developer has to insert this new curve's type. Another example in the case of a very often used specific S-shaped curve: it would be profitable to insert a new S-shaped function in the scrolling list so that the user doesn't need to set any parameters after having chosen this type but could use directly this predefined S-shape.

The fact that the generic functions are stored in the *mEdit* library does not mean that it is not possible to add new generic functions. The structure of *mEdit* is thought to allow the definition of new generic functions in a simple way. Without going into details, some modifications of the definition window (that is the `mEdit.client.DefinitionPanel` class) are required when adding a new curve's type. Regarding the layout, the label of the new type should be inserted in the scrolling list. When selecting this type, an appropriate panel should allow the user to set the parameters of the new type. The graphical components contained in this new panel can be different than sliders.

After having changed the layout, the developer needs to define the class of the new generic function. This function must be coded as a new sterilizable class extending the `mEdit.functions.Function` class like the existing `mEdit.functions.LinearFunction` and `mEdit.functions.SFunction` classes. As explained in subsection 6.1.5, the `mEdit.functions.Function` class ensures that every class extending it has the required properties of a generic function and can be properly used inside the *mEdit* editor. Then in the `actinophytoses` method of the `mEdit.client.DefinitionPanel` class, some lines of code must be inserted to instantiate this new class when the user selects the corresponding curve's type.

7.4.3. Using *mEdit* to access a membership value (for the *fcQL* interpreter)

A goal of the editor is also to provide the *fcQL* interpreter a tool to access the membership function. Since the editor is the only one which can store and retrieve the membership functions, the interpreter has to pass through the *mEdit* library to get a membership value.

As the only way in *mEdit* to access the meta-tables is the database controller, the `mEdit.database.DBController` class is the entry point. The first step is to retrieve the membership function corresponding to a specific term and a specific linguistic variable. Using the `detruncation` method of the `mEdit.database.DBController` class and passing the term and linguistic variable as parameters, the corresponding membership function will be returned as a `MFunction` object. The second step is to apply this function to an attribute's value. This can be made with the `manneristically` method of the `MFunction` class which computes the membership value of an attribute's value.

Conclusion

As the previous chapters have shown, all the goals of this work have been successfully reached. *mEdit* is able to set the membership values and to define membership functions related to respectively discrete and continuous attributes. The membership functions can be graphically and piecewise specified and the editor hides the corresponding mathematical equation whose parameters are finally stored in the meta-tables. The editor is simple to use and does not require a particular mathematical background. Specialists can easily use it but also all other kind of users. Moreover, through the database controller of *mEdit*, the *fCQL* interpreter can retrieve the membership values from the meta-tables. Finally, a special attention has been given to the correctness of the code and for that purpose the *mEdit* editor has been conscientiously tested.

8.1. Strength of *mEdit*

mEdit is a light and easy-to-use membership editor. If the software requirements for installing *mEdit* are fulfilled, it can be quickly deployed on a network: putting the servlet files in the appropriate directory, running the database script to create the meta-tables, starting the database and the servlets server and then launching the applet. Because of its Java-based architecture, it can easily be installed in every environment. The client-side applet can be used as an independent Java program or as a plug-in embedded in a HTML or a JavaServer page. The servlet-applet communication operating through simple HTML requests reduces the firewalls configuration concerns as no particular range of ports needs to be opened for socket connections. On the other hand, if the meta-information needs to be secured, *mEdit* could be quickly extended to support the HTML over SSL¹ protocol.

The main strength of *mEdit* is its graphical environment for the definition of membership functions. By setting simple and meaningful parameters, the user graphically defines the curve and all the parameter's changes are displayed in real-time on the plot. This very intuitive and powerful by its simplicity interface requires no specific knowledge. The user outlines the membership function

¹ Secure Socket Layer

without using mathematical expression. Moreover the two types of curves, linear and S-shaped, used together can generate a nearly infinite number of functions. Therefore, each human concept should find an appropriate membership function among the wide choice of functions offered by *mEdit*.

Finally, the structure of the editor allows to store every kinds of membership functions. On the one hand, the structure to create new membership functions and new generic functions is provided by the `mEdit.functions` package and the `mEdit.MFunction` class. On the other hand, every type of parameter can be stored in the `parameter` meta-table. With this architecture, it is possible to extend in a simple way the editor with other generic functions. The *mEdit* Java library has really been designed to be extensible and its structure is logic and clear.

8.2. Improvements

No outcome is perfect. For this reason, this last section proposes some functional improvements besides the usual code or protocol optimization.

The goal of the *mEdit* editor is to define and store membership values and functions related to terms. Moreover, the editor is able to create and delete terms and their definition from the relational database. A useful add-on would be to be able to create and delete linguistic variables. In this manner, *mEdit* would be a complete tool and interface for modifying the main components of the meta-tables without using *SQL* statements.

Another add-on could be implemented in the definition window. To improve the ease of use, the user should see on the plot the membership function of the other terms related to the same linguistic variable. This way, he could define the membership function of a term seeing the other related terms.

An enhancement could be imagined to ensure the safety and persistence of the generic functions. Since those models are serializable Java classes (`LinearFunction` and `SFunction`), a solution would be to store their `.class` or even `.java` files in the meta-tables. Therefore, the definition of the generic functions could be at any time retrieved and reused to instantiate new Java object. This prevents *mEdit* of losing any data and would be a way of gathering the parameters and the generic functions in the meta-tables of the database.

Bibliography

- [1] Meier Andreas. *Relationale Datenbanken - Leitfaden für die Praxis*. Springer, Berlin, 4 edition, 2001.
- [2] Guoqing Chen. *Fuzzy Logic in Data Mining. Semantics, Constraints and Database Design*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1998.
- [3] Savary Christian. Classification et requêtes aux bases de données avec la logique floue. Master's thesis, Computer Science Department, University of Fribourg, Switzerland, 2003.
- [4] Mary Dageforde. *Trail: Security in Java 2 SDK 1.2*. [online] available: <http://java.sun.com/docs/books/tutorial/security1.2/>, accessed 13th December 2003.
- [5] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, 4th edition, March 2002.
- [6] Apache Software Foundation. *The Apache Jakarta Project, Apache Tomcat Homepage*. [online] available: <http://jakarta.apache.org/tomcat/>, accessed 13th December 2003.
- [7] Free Software Foundation. *GNU General Public License*. [online] available: <http://www.gnu.org/licenses/gpl.html>, accessed 20th December 2003.
- [8] Free Software Foundation. *The GNU Lesser General Public License*. [online] available: <http://www.gnu.org/copyleft/lesser.html>, accessed 18th December 2003.
- [9] Clavel Gilles, Mirouze Nicolas, Munerot Sandrine, Pichon Emmanuel, and Soukal Mohamed. *Java La synthèse*. Dunod, Paris, 2nd edition, 2000.
- [10] Schindler Günter. *Fuzzy-Datenanalyse durch kontextbasierte Datenbankankfragen*. Deutscher Universitäts-Verlag (DUV), Wiesbaden, 1998.
- [11] Schindler Günter. Unscharfe klassifikation durch kontextbasierte datenbankankfragen. In *Informatik (Zeitschrift SVI/FSI)*, number 3, pages 18–22, 1999.

BIBLIOGRAPHY

- [12] Chen GQ. *Logic in Data Modeling - Semantics, Constraints and Database Design*. Kluwer Academic Publishers, Boston, 1998.
- [13] PostgreSQL Global Development Group. *Java/JDBC pages for PostgreSQL*. [online] available: <http://jdbc.postgresql.org/download.html>, accessed 22th December 2003.
- [14] Bergsten Hans. *JavaServer Pages*. O'Reilly, Paris, 1st edition, 2001. french edition.
- [15] Dombi J. Membership function as an evaluation. In *Fuzzy Sets and Systems*, volume 35, pages 1–21. Elsevier, 1991.
- [16] Galindo J., Cubero J.C., Pons O., and Medina J.M. A server for fuzzy sql queries. In *Proceedings of the 1998 workshop FQAS 98 (Flexible Query-Answering Systems)*, pages 161–171, Roskilde, Denmark, May 1998.
- [17] Hunter Jason and Crawford William. *Servelts Java*. O'Reilly, Paris, 2nd edition, December 2002. french edition.
- [18] Medina J.M., Vila M.A., Pons O., and Cubero J.C. Towards the implementation of a generalized fuzzy relational database model. In *Fuzzy Sets and Systems*, volume 75, pages 273–289. Elsevier, 1995.
- [19] Medina J.M., Pons O., and Vila M.A. An elemental processor of fuzzy sql. In *Mathware and Soft Computing*, volume 3, pages 285–295. 1994.
- [20] Medina J.M., Pons O., and Vila M.A. Gefred: A generalized model for fuzzy relational databases. In *Informations Sciences*, volume 77, pages 87–109. 1994.
- [21] J. Kacprzyk and S. Zadrozny. Fquery for access: Fuzzy querying for windows-based dbms. In *Fuzziness in Database Management Systems*, P. Bosc and J. Kacprzyk. Physica-Verlag, 1995.
- [22] Masao Mukaidono. *Fuzzy Logic For Beginners*. World Scientific, Paris, 2001.
- [23] Bosc P. and Pivert O. Sqlf : a relational database language for fuzzy querying. In *IEEE Transactions on fuzzy systems*, volume 3, pages 1–17. Elsevier, february 1995.
- [24] Zadrozny S. and Kacprzyk J. Implementing fuzzy querying via the internet/www : java applets, activex controls and cookies. In *Proceedings of the 1998 workshop FQAS 98 (Flexible Query-Answering Systems)*, pages 358–369, Roskilde, Denmark, May 1998.
- [25] H.R. Schwarz. *Numerische Mathematik*. Teubner, Stuttgart, 4th edition, 1997.

BIBLIOGRAPHY

- [26] Shenoj Sujeet. On classicalizing fuzzy databases. In *Proc. of Fifth IFSA World Congress*, pages 592–595, 1993.
- [27] Takahashi Y. A fuzzy query language for relational databases. In *Systems, Man and Cybernetics, IEEE Transactions*, volume 21 6, pages 1576 –1579. Nov.-Dec. 1991.
- [28] L.A. Zadeh. Fuzzy sets. In *Information Control*, volume 8, pages 338–353. 1965.
- [29] L.A. Zadeh. Pruf-a meaning representation language for natural langages. In *Intern. J. of Man-Machine Studies*, volume 10, pages 395–460. 1978.
- [30] H.-J. Zimmermann. *Fuzzy Set Theorie - and Ist Applications*. Kluwer, Boston, Dordrecht, London, 2nd edition.

fcQL syntax

<ClassificationQuery>	→	classify <Object> from <Relation> classify <Object> from <Relation> with <ClassificationCondition>
<ClassificationCondition>	→	<ClassSelection> <AttributeSelectionList>
<AttributeSelectionList>	→	(<AttributeSelection>) <AttributeSelectionList> or (<AttributeSelection>)
<AttributeSelection>	→	<AttributeConditionList>
<AttributeConditionList>	→	<AttributeCondition> <AttributeConditionList> and <AttributeCondition>
<AttributeCondition>	→	<Attribute> is <EquivalenceClass> (<Attribute> is <EquivalenceClass> or <EquivalenceClassList>)
<EquivalenceClassList>	→	<EquivalenceClass> <EquivalenceClassList> or <EquivalenceClass>
<Attribute>	→	ColumnDefinition
<EquivalenceClass>	→	ColumnDefinition
<Relation>	→	RelationIdentifier ViewIdentifier
<ClassSelection>	→	<ClassConditionList>
<ClassConditionList>	→	<ClassCondition> <ClassConditionList> or <ClassCondition>
<ClassCondition>	→	class is <Description>
<Description>	→	ColumnDefinition
<Object>	→	ColumnDefinition

SQL commands creating the meta-tables

```
CREATE TABLE tables (  
    id                INTEGER,  
    name              VARCHAR(40)    NOT NULL,  
    PRIMARY KEY (id)  
);  
  
CREATE TABLE attribute_type (  
    id                SMALLINT        NOT NULL,  
    name              VARCHAR(40)    NOT NULL,  
    PRIMARY KEY (id)  
);  
  
CREATE TABLE linguistic_variable (  
    id                SERIAL,  
    name              VARCHAR(40)    NOT NULL,  
    attribute_name    VARCHAR(40)    NOT NULL,  
    attribute_type_id SMALLINT        NOT NULL,  
    table_id          INTEGER        NOT NULL,  
    domain_start     NUMERIC        NULL,  
    domain_end       NUMERIC        NULL,  
    PRIMARY KEY (id),  
    FOREIGN KEY (attribute_type_id) REFERENCES attribute_type (id),  
    FOREIGN KEY (table_id) REFERENCES tables (id)  
);  
  
CREATE TABLE term (  
    id                SERIAL,  
    name              VARCHAR(40)    NOT NULL,  
    linguistic_variable_id INTEGER    NOT NULL,  
    PRIMARY KEY (id),  
    FOREIGN KEY (linguistic_variable_id) REFERENCES  
        linguistic_variable (id) ON DELETE CASCADE  
);
```

APPENDIX B. SQL COMMANDS CREATING THE META-TABLES

```
CREATE TABLE category (  
    id SERIAL,  
    term_id INTEGER NOT NULL,  
    name VARCHAR(40) NOT NULL,  
    value NUMERIC NOT NULL,  
    PRIMARY KEY (id),  
    FOREIGN KEY (term_id) REFERENCES term (id)  
        ON DELETE CASCADE  
);
```

```
CREATE TABLE function (  
    id SERIAL,  
    term_id INTEGER NOT NULL,  
    function_name VARCHAR(40) NOT NULL,  
    function_type SMALLINT NOT NULL,  
    interval_min NUMERIC NOT NULL,  
    min_included BOOLEAN NOT NULL,  
    interval_max NUMERIC NOT NULL,  
    max_included BOOLEAN NOT NULL,  
    PRIMARY KEY (id),  
    FOREIGN KEY (term_id) REFERENCES term (id)  
        ON DELETE CASCADE,  
    CHECK (interval_min < interval_max)  
);
```

```
CREATE TABLE parameter (  
    id SERIAL,  
    function_id INTEGER NOT NULL,  
    name VARCHAR(40) NOT NULL,  
    value NUMERIC NOT NULL,  
    PRIMARY KEY (id),  
    FOREIGN KEY (function_id) REFERENCES function (id)  
        ON DELETE CASCADE  
);
```

***mEdit* installation CD-ROM**
