A Primer in Fuzzy Markup Language ver. 0.1.1

Giovanni Acampora[1]
Autilia Vitiello[2]

[1]School of Science and Technology,
Nottingham Trent University,
Nottingham NG11 8NS, UK
[2]Department of Mathematics and Computer Science
University of Salerno
Fisciano, 84084, Italy

Introduction

Fuzzy control theory can be considered as the most widely used application of fuzzy logic. From a high level point of view, a Fuzzy Logic Controller (FLC) is an adequate methodology for designing and developing controllers capable of supplying high quality performance in environments characterized by high level of uncertainty and imprecision. Moreover, FLCs lets controller designers to describe complex systems using their knowledge and experience by means of linguistic IF-THEN rules. It does not require any system modeling or complex math equations governing the relationship between inputs and outputs as it happens for others controller design methodologies (e.g. PID). FLC typically takes only a few rules to describe systems that may require several of lines of conventional software. As a result, fuzzy logic significantly simplifies controller design complexity. However, in spite of these unquestionable advantages, *the real design of FLCs is strongly depends upon hardware architecture which will implement the running version of designed controller*. *For this reason, a new version of FLCs implementation allowing the designer to model the controllers in hardware independent way has been considered*.

This novel vision of FLCs is based on the labeled tree idea, a data structure defined by means of the well-know graph theory. Because the labeled trees are data models derived by the XML-based document representation, each fuzzy controller is representable by means of XML, the main technology for data abstraction. So, the development of a new XML-based language able to model a fuzzy controller enable designers to represent controllers in a human-readable and hardware independent way. This new language is named Fuzzy Markup Language (FML).

FML is essentially composed by three layers:

1) XML in order to create a new markup language for fuzzy logic control;
2) a document type definition (DTD), initially, and now a XML Schema in order to define the legal building blocks; and
3) extensible stylesheet language transformations (XSLT) in order to convert a fuzzy controller description into a specific programming language.

In order to model a fuzzy system by means of XML it is necessary to analyzed a controller by a structural point of view. The high-level structure of fuzzy controller is shown in Fig. 1.
The main components of fuzzy controller are:

- fuzzy knowledge base;
- fuzzy rule base;
- inference engine
- fuzzification subsystem;
- defuzzification subsystem.

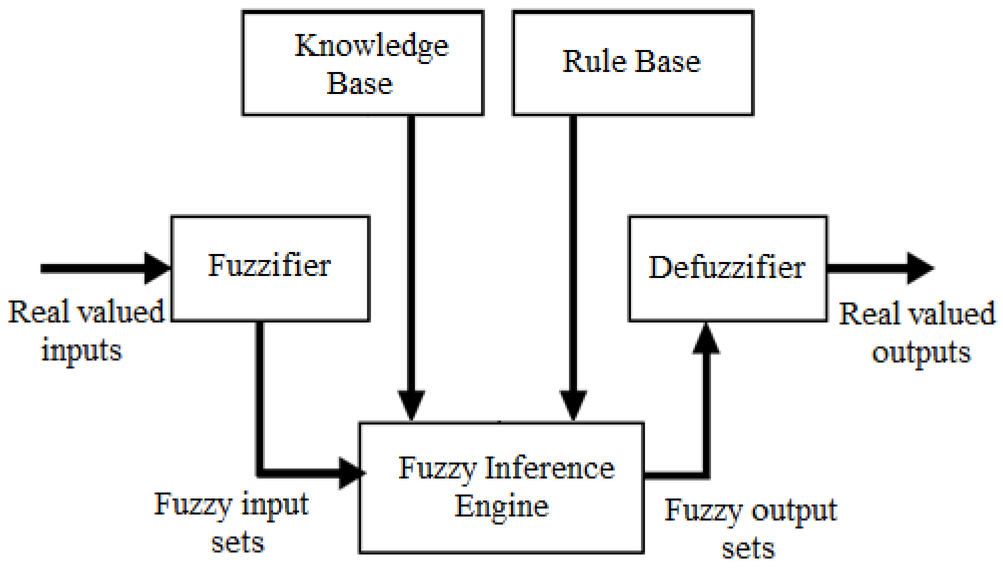


Fig. 1: Generic fuzzy control

The fuzzy knowledge base manipulates the variables used in the controlled system (such as *temperature*, *pressure*, etc.), corresponding to the knowledge used by human experts. The Fuzzy Rule Base represents the set of relations between fuzzy variable defined in the controller system. The Inference Engine is the fuzzy controller component able to extract new knowledge from fuzzy knowledge base and fuzzy rule base. Moreover, the controlled system works with real numbers, whereas the fuzzy controller system works with fuzzy concepts. The two subsystems, the fuzzification subsystem and the defuzzification subsystem, are necessary to bridge controlled systems with controller systems. The former permits to transform the real numbers used by controlled systems into a fuzzy set used by fuzzy controller. The latter transforms the fuzzy set generated by fuzzy controller into real numbers usable by controlled system. The nature of consequent part of fuzzy rules permits to define two kind of fuzzy controller: the Mamdani controller and the Takagi-Sugeno-Kang (TSK) controller. The Mamdani controller uses a fuzzy set to model the consequent part of rule, whereas the TSK controller uses the linear function of input variable to describe the rule consequent part. FML permits to model both fuzzy controllers. For sake of simplicity, and to better introduce the operating concepts, the discussion is focused on a Mamdani fuzzy controller. Then, we will describe also a Takagi-Sugeno-Kang system to show how FML manages this kind of controller. The described system is useful to regulate the tipping in, for example, a restaurant. It has got two variables in input (*food* and *service*) and one in output (*tip*).

However, FML uses an alternative representation of fuzzy controller in order to derive an XML-based model. This representation is based on the notion of the labeled tree (Fig. 2). Each node in fuzzy controller tree will represent an XML tag, and the father-child relation represents a nested relation between related tags.

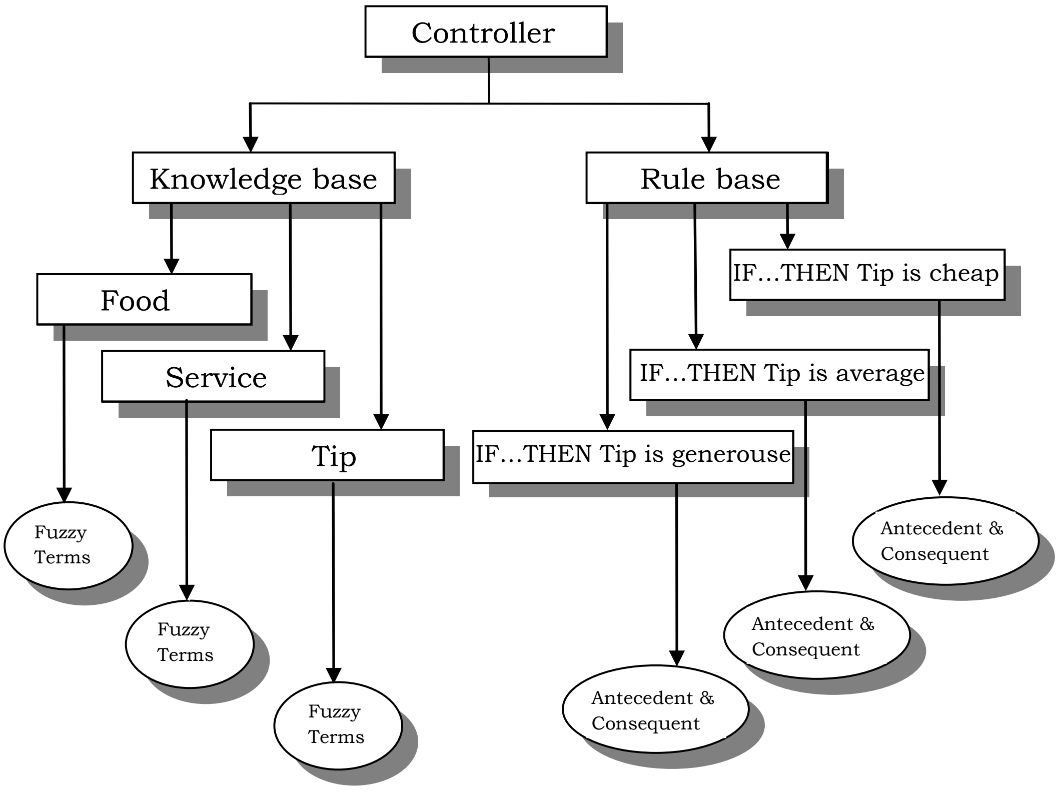


Fig. 2: Fuzzy controller tree

Starting from Fig. 2 it is simple to individuate the collection of XML tags capable of modeling a fuzzy controller. In detail, the controller node can be modeled by means of a tag named ***<FUZZYCONTROLLER>***. Such tag represents the root tag of FML program, that is, the opening tag of each FML program. <FUZZYCONTROLLER> has two attributes: *name* and *ip*. The *name* attribute permits to specify the name of fuzzy controller and *ip* is used to define the location of controller in the computer network. Considering the fuzzy left subtree the knowledge base component is encountered. The fuzzy knowledge base is defined by means of the tag ***<KNOWLEDGEBASE>*** which maintains the set of fuzzy concepts used to model the fuzzy rule base. The <KNOWLEDGEBASE> uses the attribute *ip* that determines the location in the network of whole fuzzy knowledge base of our system.
In order to define the fuzzy concept related controlled system, <KNOWLEDGEBASE>
tag uses a set of nested tags:
• ***<FUZZYVARIABLE>*** defines the fuzzy concept, for example, "*food"*;
• ***<FUZZYTERM>*** defines a linguistic term describing the fuzzy concept, for example, "*rancid food"*;
• a set of tags defining a shape of fuzzy sets are related to fuzzy terms.
The attributes of <FUZZYVARIABLE> tag are: *name*, *scale*, *domainLeft*, *domainRight*, *type* and, for only an output, *accumulation*, *defuzzifier* and *defaultValue*. The *name* attribute defines the name of fuzzy concept, for instance, *food*; *scale* is used to define the scale used to measure the fuzzy concept, for instance, *Celsius degree*, instead, in the our case, for variable *food* it can be set to null; *domainLeft* and *domainRight* are used to model the universe of discourse of fuzzy concept, that is, the set of real values related to fuzzy concept, for instance [0°,40°] in the case of Celsius degree or [0,10] for variable food to indicate that food can be judge with a value from 0 to 10; the position of fuzzy concept into rule (consequent part or antecedent part) is defined by *type* attribute (input/output); *accumulation* attribute defines the method of accumulation that is a method that permits the combination of results of a variable of each rule in a final result; *defuzzifier* attribute defines the method used to execute the conversion from a fuzzy set, obtained after aggregation

process, into a numerical value to give it in output to system; *defaultValue* attribute defines a real value used only when no rule has fired for the variable at issue.
<FUZZYTERM> uses two attributes: *name* used to identify the linguistic value associate with fuzzy concept and *complement*, a boolean attribute that defines, if it is true, it is necessary to consider the complement of membership function defined by given parameters.
Fuzzy shape tags, used to complete the definition of fuzzy concept, are:
<TRIANGULARSHAPE>
<RIGHTLINEARSHAPE>
<LEFTLINEARSHAPE>
<PISHAPE>
<GAUSSIANSHAPE>
<RIGHTGAUSSIANSHAPE>
<LEFTGAUSSIANSHAPE>
<TRAPEZOIDSHAPE>
<SSHAPE>
<ZSHAPE>
<RECTANGULARSHAPE>
<SINGLETONSHAPE>
Every shaping tag uses a set of attributes which defines the real outline of corresponding fuzzy set. The number of these attributes depends on the chosen fuzzy set shape. Considering as example, the "*food*" as input variable and "*tip*" as output one in defining our fuzzy control, the knowledge base, and in particular the quality of food can be modeled as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<FuzzyController name="newSystem" ip="127.0.0.1">
    <KnowledgeBase>
        <FuzzyVariable name="food" domainleft="0.0" domainright="10.0" scale=""
                       type="input">
            <FuzzyTerm name="delicius" complement="false">
                <LeftLinearShape Param1="5.5" Param2="10.0"/>
            </FuzzyTerm>
            <FuzzyTerm name="rancid" complement="false">
                <TriangularShape Param1="0.0" Param2="2.0" Param3="5.5"/>
            </FuzzyTerm>
        </FuzzyVariable>
         …
        <FuzzyVariable name="tip" domainleft="0.0" domainright="20.0"
                       scale="Euro" defaultValue="0.0" defuzzifier="COG"
                       accumulation="MAX"  type="output">
            <FuzzyTerm name="average" complement="false">
                <TriangularShape Param1="5.0" Param2="10.0" Param3="15.0"/>
            </FuzzyTerm>
            <FuzzyTerm name="cheap" complement="false">
                <TriangularShape Param1="0.0" Param2="5.0" Param3="10.0"/>
            </FuzzyTerm>
            <FuzzyTerm name="generouse" complement="false">
                <TriangularShape Param1="10.0" Param2="15.0" Param3="20.0"/>
            </FuzzyTerm>
        </FuzzyVariable>
      <KnowledgeBase>
    ...
</FuzzyController>
```

A special tag that can furthermore be used to define a fuzzy shape is ***<USERSHAPE>***. This tag is used to customize fuzzy shape (custom shape). The custom shape modeling is performed via a set of ***<POINT>*** tags that lists the extreme points of geometric area defining the custom fuzzy shape. Obviously, the attributes used in <POINT> tag are x and y coordinates.
The fuzzy right subtree is used to define the rule base set. Indeed, it is possible to define more rule bases to describe the different behaviors of system. The root of this rule base is modeled by ***<RULEBASE>*** tag which defines a fuzzy rule set. The <RULEBASE> tag uses five attributes: *name*, *type*, *activationMethod*, *andMethod* and *orMethod*. Obviously, the *name* attribute uniquely identifies the rule base. The *type* attribute permits to specify the kind of fuzzy controller (Mamdani or TSK) respect to the rule base at issue. The *activationMethod* attribute defines the method used to implication process; the *andMethod* and *orMethod* attribute define, respectively, the *and* and *or* algorithm to use by default. In order to define the single rule the ***<RULE>*** tag is used. The attributes used by the <RULE> tag are: *name*, *connector*, *operator* and *weight*. The *name* attribute permits to identify the rule; *connector* is used to define the logical operator used to connect the different clauses in antecedent part (and/or); *operator* defines the algorithm to use for chosen connector; *weight* defines the importance of rule during inference engine time. The definition of antecedent and consequent rule part is obtained by using ***<ANTECEDENT>*** and ***<CONSEQUENT>*** tags. ***<CLAUSE>*** tag is used to model the fuzzy clauses in antecedent and consequent part. This tag use the attribute *modifier* to describe a modification to term used in the clause. The possible values for this attribute are: above, below, extremely, intensify, more or less, norm, not, plus, slightly, somewhat, very, none. To complete the definition of fuzzy clause the nested **<VARIABLE>** and **<TERM>** tag have to be used. A sequence of <RULE> tags realizes a fuzzy rule base.
As example, let us consider a Mamdani rule composed by *(food is rancid) OR (service is very poor)* as antecedent and *tip is cheap* as consequent. The antecedent part is formed by two clauses: *(food is rancid)* and *(service is poor)*. The first antecedent clause uses *food* as variable and *rancid* as fuzzy term, whereas, the second antecedent clause uses service as a variable, *poor* as fuzzy term and *very* as modifier; the consequent clause uses *tip* as a fuzzy variable and *cheap* as a fuzzy term. The complete rule is:
IF (*food is rancid*) **OR** (*service is very poor*) **THEN** (tip *is cheap*).

Let us see how fml language define a rule base with this rule.

```
<RuleBase name="Rulebase1" activationMethod="MIN" andMethod="MIN" orMethod="MAX"
          type="mamdani">
      <Rule name="reg1" connector="or" operator="MAX" weight="1.0">
            <Antecedent>
                <Clause>
                    <Variable>food</Variable>
                    <Term>rancid</Term>
                </Clause>
                <Clause modifier="very">
                    <Variable>service</Variable>
                    <Term>poor</Term>
                </Clause>
            </Antecedent>
            <Consequent>
                <Clause>
                    <Variable>tip</Variable>
                    <Term>cheap</Term>
                </Clause>
            </Consequent>
        </Rule>
         …
</RuleBase>
```

Now, let us see a Takagi-Sugeno-Kang system that regulates the same issue.
The most important difference with Mamdani system is the definition of a different output variable "*tip*". The ***<TSKVARIABLE>*** tag is used to define an output variable that can be used in a rule of a Tsk system. This tag has the same attributes of a mamdani output variable except for the domainleft and domainright attribute because a variable of this kind (called tsk-variable) hasn't a universe of discourse. The nested ***<TSKTERM>*** tag represents a linear function and so it is completely different from *<FUZZYTERM>*. The ***<TSKVALUE>*** tag is used to define the coefficients of linear function.
The following crunch of fml code shows the definition of output variable "tip" in a Tsk system.

```
<?xml version="1.0" encoding="UTF-8"?>
<FuzzyController name="newSystem" ip="127.0.0.1">
    <KnowledgeBase>
        …
             …
        <TSKVariable name="tip" scale="null" accumulation="MAX" defuzzifier="WA"
                     type="output">
            <TSKTerm name="average" order="0">
                <TSKValue>1.6</TSKValue>
            </TSKTerm>
            <TSKTerm name="cheap" order="1">
                <TSKValue>1.9</TSKValue>
                <TSKValue>5.6</TSKValue>
                <TSKValue>6.0</TSKValue>
            </TSKTerm>
            <TSKTerm name="generouse" order="1">
                <TSKValue>0.6</TSKValue>
                <TSKValue>1.3</TSKValue>
                <TSKValue>1.0</TSKValue>
            </TSKTerm>
        </TSKVariable>
      <KnowledgeBase>
    ...
</FuzzyController>
```

The fml definition of rule base component in a Tsk system doesn't change a lot. The only different thing is that the <CLAUSE> tag doesn't have the modifier attribute.
As example, let us consider a tsk rule composed by *(food is rancid) OR (service is very poor)* as antecedent and, as consequent, *tip=1.9+5.6*food+6.0*service* that can be written as *tip is cheap* in an implicitly way. So the rule can be written in this way:

IF (*food is rancid*) **OR** (*service is very poor*) **THEN** (tip *is cheap*).

Let us see how fml language define a rule base with this rule.

```
<RuleBase name="Rulebase1" activationMethod="MIN" andMethod="MIN" orMethod="MAX"
          type="tsk">
      <Rule name="reg1" connector="or" operator="MAX" weight="1.0">
            <Antecedent>
                <Clause>
                    <Variable>food</Variable>
                    <Term>rancid</Term>
                </Clause>
                <Clause>
                    <Variable>service</Variable>
                    <Term>poor</Term>
                </Clause>
            </Antecedent>
            <Consequent>
                <Clause>
                    <Variable>tip</Variable>
                    <Term>cheap</Term>
                </Clause>
            </Consequent>
        </Rule>
         …
</RuleBase>
```

FML has been designed to be simply embedded into distributed artificial intelligence environment and, for this reason, some tags use an additional, not fuzzy logic oriented, attribute exploited to move the different components of a controller on different computational hosts in order to achieve a massive parallelization during fuzzy inference. The additional attribute is named *ip* and it is used by the following tags: <KNOWLEDGEBASE>, <FUZZYVARIABLE>, <RULEBASE> and <RULE>.

An Overview of VisualFMLTool 0.1.1

VisualFMLTool is a development environment for fuzzy-inference-based systems. Its functionalities cover the different stages of the fuzzy system design process, from their initial description to the final implementation. It admits to designing both Mamdani and Takagi-Sugeno-Kang systems. It allows to develop complex systems and above all to manage more ones in the same time. The environment has been completely programmed in Java, so it can be executed on any platform with JRE (Java Runtime Environment) installed.
The main functionalities the tool provides are the following ones:

- Fuzzy system description: the tool allows to define the system creating linguistic variables and rules. Above all, the rules can be organized in rule bases, each one describes a particular system behavior;
- Verification: the tool allows to monitor the system behavior showing outputs corresponding to different inserted inputs and also it allows to represent graphically the system behavior showing the control surface;
- Tuning: the tool allows to apply algorithms based on neural network and not only ones to execute the learning stage and specifically to modify membership functions using a training data file (under development);
- Synthesis: the tool allows to generate high-level languages descriptions even if at the moment it provides only java implementation (under development).

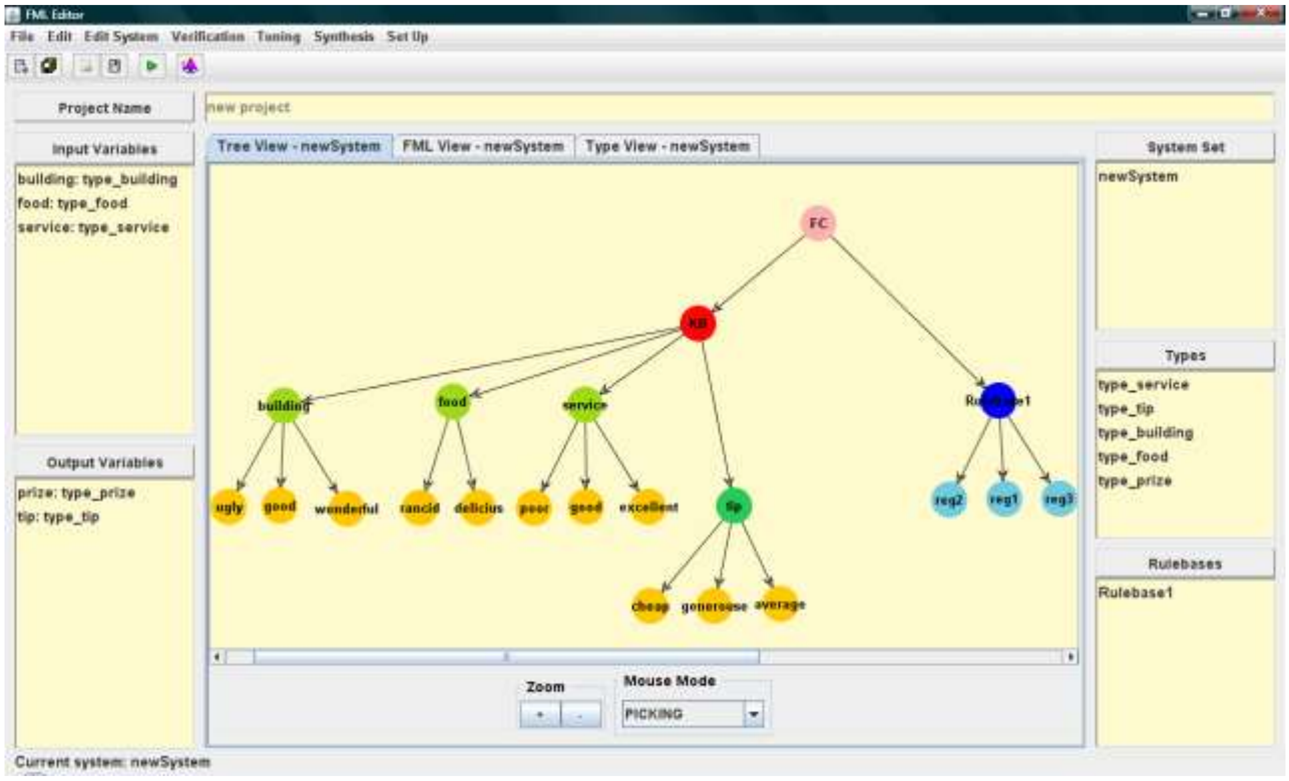
VisualFMLTool defines the system structure using FML (Fuzzy mark-up language) as specification language. Indeed it is possible to import a fml file to build a fuzzy controller, and also to export a system created by the tool in a fml file.

Installation of VisualFMLTool 0.1.1

VisualFMLTool can be executed on platforms containing the Java Runtime Environment. The Java Software Development Kit, including JRE, compiler and many other tools can be found at http://java.sun.com/j2se/.
To install VisualFMLTool is needed to download the visualFMLTool.zip from web site http://www.di.unisa.it/dottorandi/avitiello/FML/VisualFMLTool-0.1.1.zip and to extract it. Then it is only needed to click the file visualFMLTool.bat included in the zip to execute the tool.

VisualFMLTool: description

The next figure shows the main window of VisualFMLTool.



Project Name: new project

Input Variables
building: type_building
food: type_food
service: type_service

Output Variables
prize: type_prize
tip: type_tip

System Set
newSystem

Types
type_service
type_tip
type_building
type_food
type_prize

Rulebases
Rulebase1

Mouse Mode: PICKING

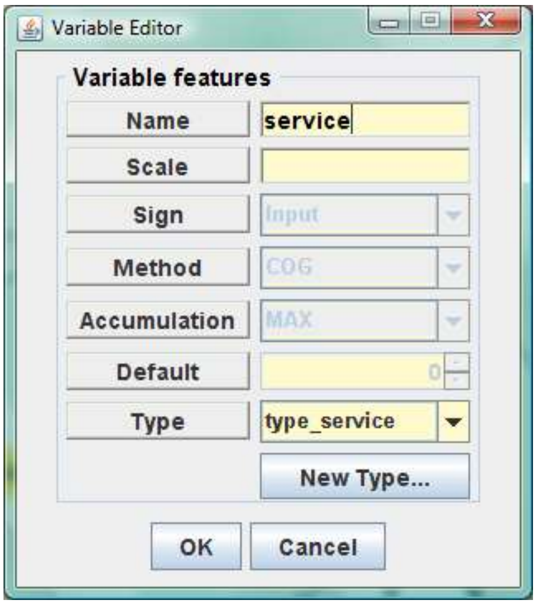
Current system: newSystem

Fig. 3: Initial Frame

The menu bar in the main window permits to execute all the functionalities; instead the toolbar under it contains only the most used options. Under the toolbar there is a field to view the project name, this field is not editable. The name of the project under development can be changed by the "*Save Project As"* option. The central zone is divided into three parts. The left and the right parts contain five lists: two ones on the left side that show input and output variables of system and three ones on the right side that show, beginning from top, the systems loaded or created by tool, types of variables, the rule bases. In the central area you can switch from a "*Tree view*" that shows a graph representing the structure of system according to fml specifications, to a "*FML view*" that shows fml code or to a "*Type view*" that shows a graph for each root type representing the hierarchy of root type. At the bottom, the "*Tree view*" and "*Type view*" has got two buttons "+" and "-" to zoom graphs and a pulldown list with two item "Transforming" and "Picking". The first one permits to move the whole graph on the view area, instead, the second one admits to moving each node of graph on the view area. The *File* menu allows to create, load, save and close a project, that can contain one or more systems. The menu ends with the option to exit the environment. The *Edit* menu allows to create, load, save, delete a fuzzy system, import a file fml to build automatically a system, export the current system in a file fml or the graph as a png file, and finally to switch from a system to another. The *Edit System* menu allows to edit the current system, that is the system selected in system list shown on the right side. The possible actions are creating, modifying and deleting a variable or a type or a rule base of system. The *Verification* menu allows to represent the system behavior on a 3-dimensional plot and monitoring the system. The *Tuning* menu allows to start learning algorithms (under development). The *Synthesis* menu allows the software synthesis, that generates system description in Java (under development). The *Set Up* menu is used to modify the environment changing its the look and feel. Many options on the menu bar are only enabled when a fuzzy system is selected.

Fuzzy System Description

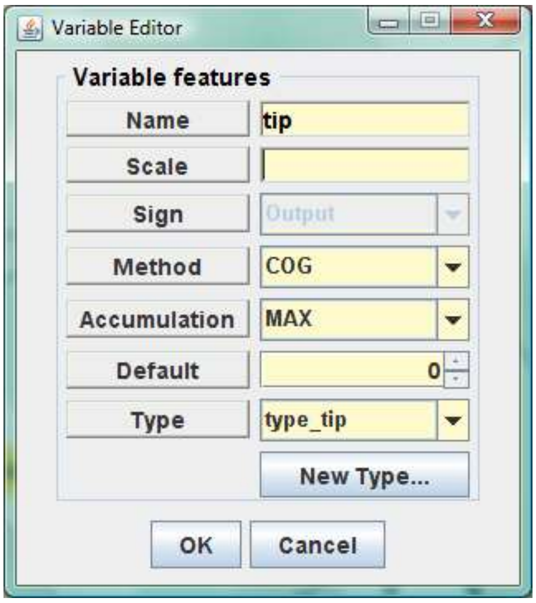
VisualFMLTool offers a graphical interface to ease the description of fuzzy systems, avoiding the need for an in depth knowledge of the FML language. This stage of fuzzy process is covered by actions of “*Edit”* and “*Edit System”* menus. Besides a lot of these actions are admitted also by means of popups shown clicking the lists on the left and right side using the right button of mouse or clicking on a graph node in the “*Tree view*” area.
The first step in the description of a fuzzy system is creating input and output linguistic variables, by means of the *Variable Editor* window shown below.


Variable Editor

Variable features

Name	service
Scale	
Sign	Input
Method	COG
Accumulation	MAX
Default	0
Type	type_service

Fig. 4: Variable Editor - input variable

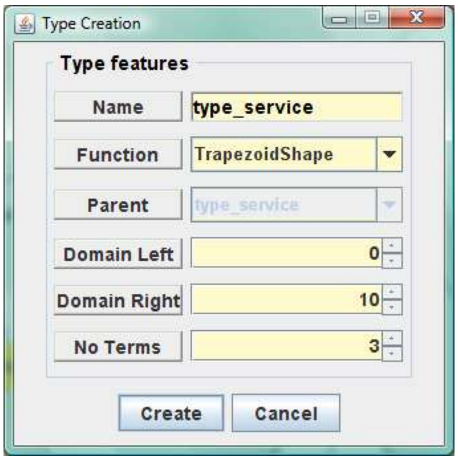

Variable Editor

Variable features

Name	tip
Scale	
Sign	Output
Method	COG
Accumulation	MAX
Default	0
Type	type_tip

Fig. 5: Variable Editor - output variable

A new variable needs the introduction of its name, its scale and its sign (input/output). The defuzzification method, the accumulation one and the default value are necessary only for output variables and so they are disabled for an input variable. Besides each variable must have a type, so you can create it beginning from this window clicking “*New Type*” or before creating the variable clicking on “*New Type*” from menu “*Edit System*” or from popup shown when you click on type list. The tool manages two kind of types: one that we call simply *type* that can be associated to an input variable of a Mamdani or Takagi-Sugeno-Kang system and to an output variable only of a Mamdani system and another, that we call *tsk-type*, that can be associated only to an output variable of a Takagi-Sugeno-Kang system. We use the word type not in italics to refer a type in a generic way; that is to refer both types.
Once a variable has been created it can be modified in all attributes except for its sign. To modify a variable you can select it on input or output list and then double click or you can click right button of mouse on the same lists and select the item “*Modify variable*” on popup shown or finally you can modify a variable by means of popup shown clicking on a variable node of graph. The functionality “*Delete variable*” is reachable in the same way.
The fig.6 shows the window to create a *type*, whereas the fig.10 shows that one to create a *tsk-type*.



Type Creation

Type features

Name	type_service
Function	TrapezoidShape
Parent	type_service
Domain Left	0
Domain Right	10
No Terms	3

Create Cancel

Fig. 6: Type Creation

A new *type* needs the introduction of its identifier, universe of discourse (domain left and domain right) and an initial number of terms (it can be modified adding or deleting a term by means of option “*Modify*” in “*Edit System*” menu or in popup). The window includes several predefined functions *c*orresponding to the most usual partitions of the universes. These predefined functions contain triangular, trapezoidal, rectangular, singleton, gaussian and linear partitions. Besides they include the “*User shape*” option, which admits to defining a custom function inserting all points of membership function. Finally, the “*EXTENDS*” option admits the extension of an existing *type* (selected in the *Parent* field). This concept (extension of a type) isn’t implemented in the FML language, but it is provided by tool to make fuzzy design more flexible. The rules of extension of a *type* are:

- The child-type inherits automatically the universe of discourse and the terms of its parent.
- The child-type can add a term or overwrite a term of parent.
- The child-type can’t delete a term that belongs to its parent without causing the cancellation of term also in its parent.

The modifications between a child-type and a parent-type can be noticed also by means of a tooltip shown when the mouse is on an edge of graph of a root type.
The next figure shows an example.

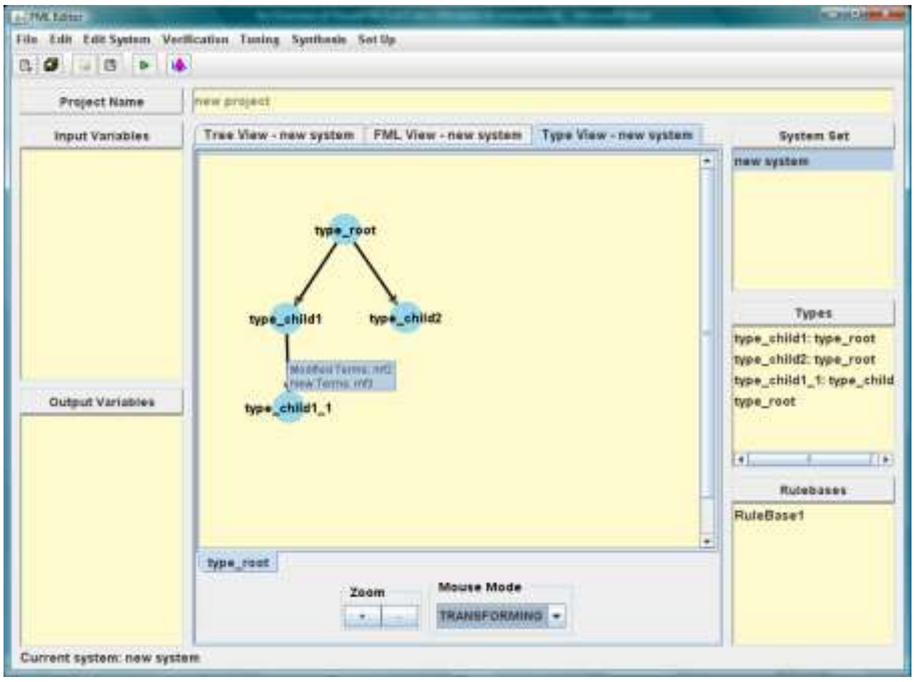
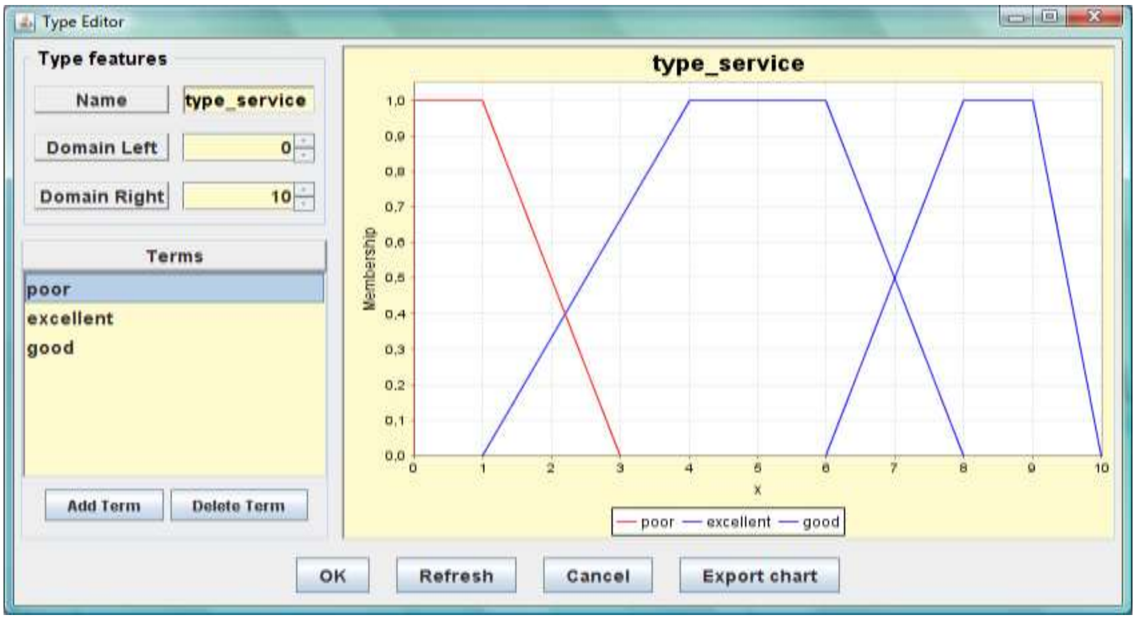


Fig. 7: Type hierarchy

Once a *type* has been created, it can be edited using the *Type Editor* window shown below.

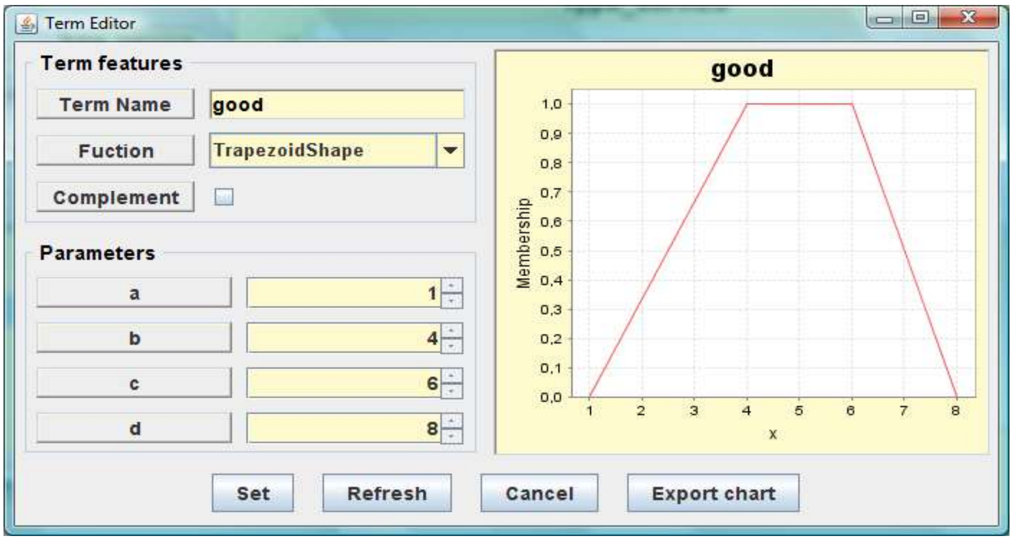


Name: type_service
Domain Left: 0
Domain Right: 10

Terms:
poor
excellent
good

Fig. 8: Type Editor

This window allows the modification of the *type* name and universe of discourse. Besides it admits to adding, editing and removing the membership functions(terms) of the edited *type*. The window shows a graphical representation of the membership functions, where the selected membership function is represented in a different color (red). The bottom of the window presents a command bar with the usual buttons to save or reject the last changes, and to refresh modifications before saving them. Besides the "*Export chart*" button allows to save the graphical representation as a jpeg file. It is worth considering that the modifications on the definition of the universe of discourse can affect the membership functions already defined. Hence, a validation of the membership function parameters is done before saving the modifications, and if a membership function definition becomes invalid it is updated in a default way (it is preferable to view only an error message). A membership function can be created or deleted with buttons under terms list and edited with double click on the terms list.

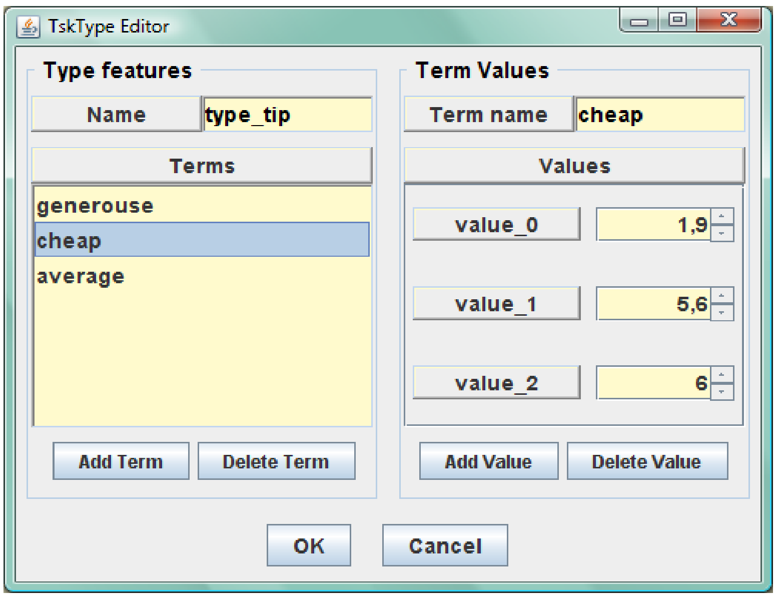


Term Name: good
Fuction: TrapezoidShape
Complement: ☐

a: 1
b: 4
c: 6
d: 8

Fig. 9: Term Editor

The previous figure shows the window to edit a membership function (term). The window has fields to introduce the name, to select the kind of membership function, and to introduce the parameter values. The right side of the window shows a graphical representation of membership function. The bottom of the window shows a command bar with four options: *Set*, to close the window saving the changes, *Refresh*, to repaint the graphical representation, *Export chart*, to save the graphical representation as a jpeg file, and *Cancel*, to close the window without saving the modifications.



Name: type_tip

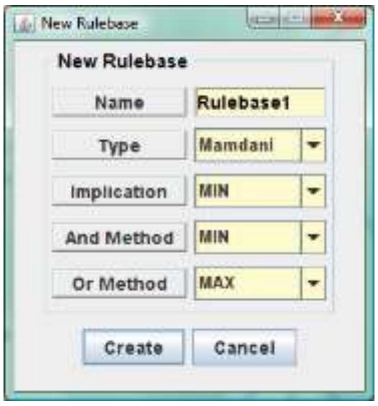
Terms:
- generouse
- cheap
- average

Term name: cheap

value_0	1,9
value_1	5,6
value_2	6

Fig. 10: Tsk-type Editor

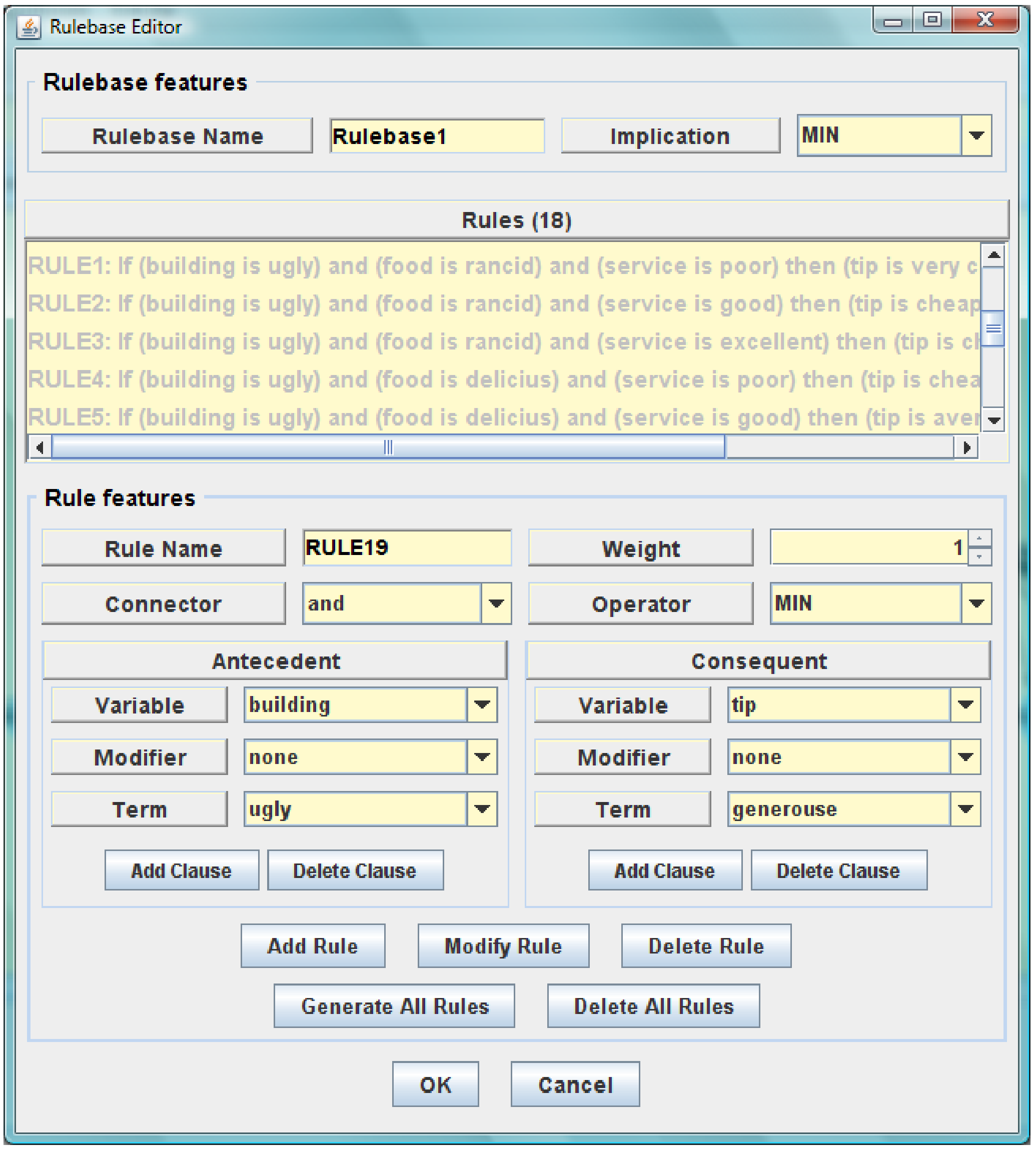
A *tsk-type* needs the introduction of a name and it has an only term by default. Then you can add, edit or delete a term. Each term has an only value by default, but then you can add, edit or delete one. A tsk-variable hasn't got a universe of discourse. A *tsk-type* can't be extended. All *tsk-types* are shown in an only graph.
To modify a type you can select it on type list and then double click or you can click right button of mouse on the same list and select the item "*Modify type*". The functionality "*Delete type*" is reachable in the same way. When you modify a type, automatically you modify all variables associated to it. When you delete a type that is already associated to a variable you delete automatically also the variable. If you don't want to delete the variable you must associate to it another type before deleting the associated type.
The next step in the definition of a fuzzy system is to describe the rule bases expressing the relationship among the system variables. Rule bases can be created, edited and removed from their list by means of a popup or from "*Edit System*" menu. The "*New Rulebase*" window is used to create a rule base.



Name: Rulebase1
Type: Mamdani
Implication: MIN
And Method: MIN
Or Method: MAX

Fig. 11: New Rulebase

A rule base needs the introduction of a name, an activation method, a type (Mamdani/Tsk), an operator by default for "and connector" and "or connector". When you click on "Create" button, according to type of rule base, you view the window "*Rulebase Editor*" (fig.12) or "*TskRulebase Editor*" (fig.13). Both windows try to ease the rule base design.



Rulebase Name: Rulebase1
Implication: MIN

RULE1: If (building is ugly) and (food is rancid) and (service is poor) then (tip is very c
RULE2: If (building is ugly) and (food is rancid) and (service is good) then (tip is cheap
RULE3: If (building is ugly) and (food is rancid) and (service is excellent) then (tip is c
RULE4: If (building is ugly) and (food is delicius) and (service is poor) then (tip is chea
RULE5: If (building is ugly) and (food is delicius) and (service is good) then (tip is aver

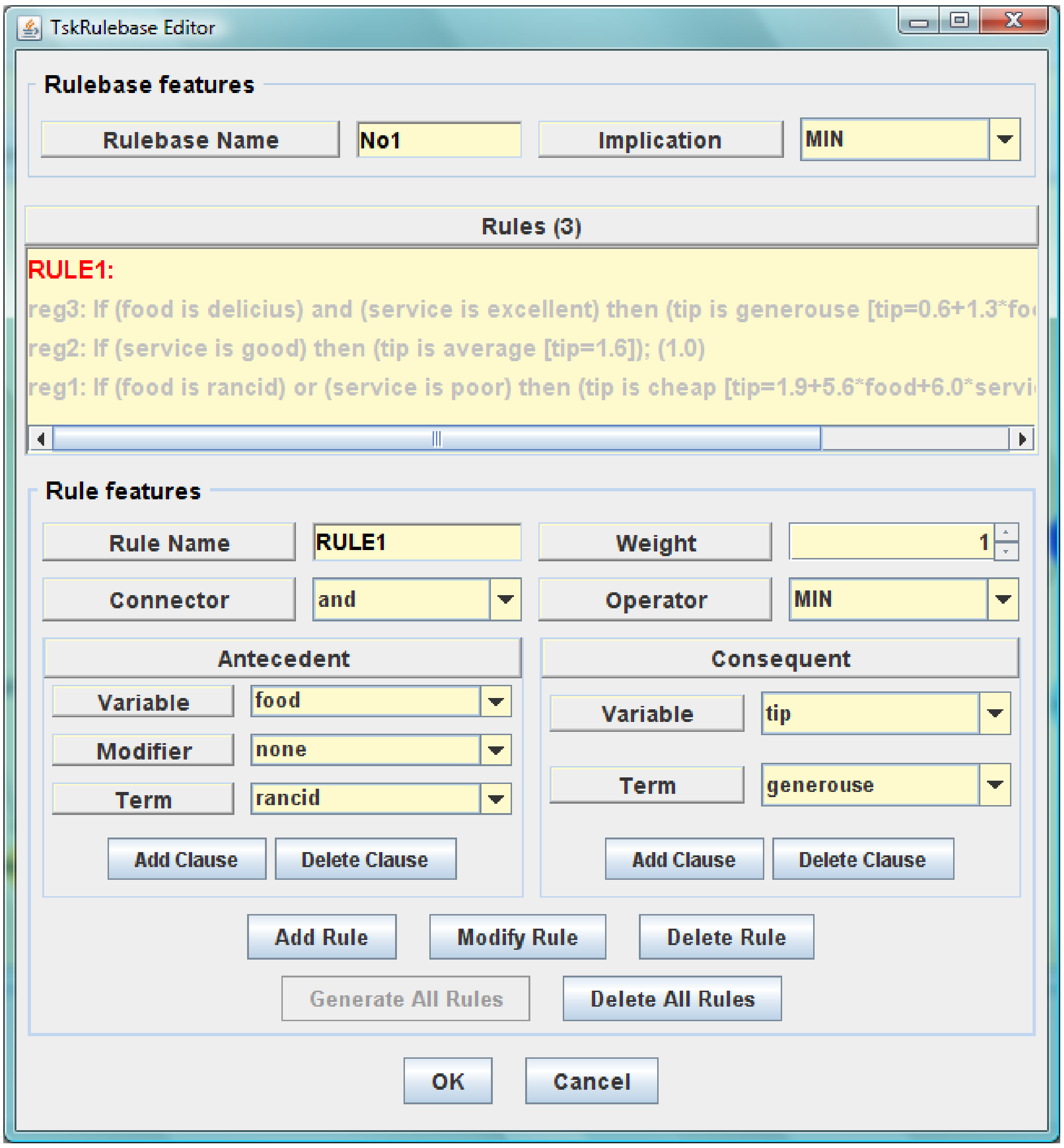
Rule Name: RULE19
Weight: 1
Connector: and
Operator: MIN

Antecedent — Variable: building; Modifier: none; Term: ugly
Consequent — Variable: tip; Modifier: none; Term: generouse

Fig. 12: Rulebase Editor

The previous window is divided into four zones: at the top there are the fields to introduce the name of the rule base and the activation method; the central zone is dedicated to showing the contents of the rules included in the rule base; under rules list there are all components necessary to build a new rule and the bottom part of the window contains the command bar with the usual buttons to save or reject the modifications. The current rule is the only rule with red color. When the current rule is built you can add it to the rule base by means of button "*Add Rule*". Each rule can be modified or deleted using buttons "*Modify Rule*" and "*Delete Rule*" after selecting it. In order to generate all possible rules you can use the bottom "*Generate All Rules*", instead, if you want to delete all rules you can use the bottom "*Delete All Rules*". The output variables that can be selected in the consequent part are only variables associated with a *type*, instead, they are variables associated with a t*sk-type* when you are building a tsk-rule base.

The next figure shows a "*TskRulebase Editor*". The window structure of "*TskRulebase Editor*" and "*Rulebase Editor*" is almost the same except that, in a tsk-rule base, the consequent part can't have got a modifier. It is worth marking as a tsk-rule is written, that is you explicitly can see the linear function associated to an output variable.



Rulebase Name: No1
Implication: MIN

RULE1:
reg3: If (food is delicius) and (service is excellent) then (tip is generorse [tip=0.6+1.3*fo
reg2: If (service is good) then (tip is average [tip=1.6]); (1.0)
reg1: If (food is rancid) or (service is poor) then (tip is cheap [tip=1.9+5.6*food+6.0*servi

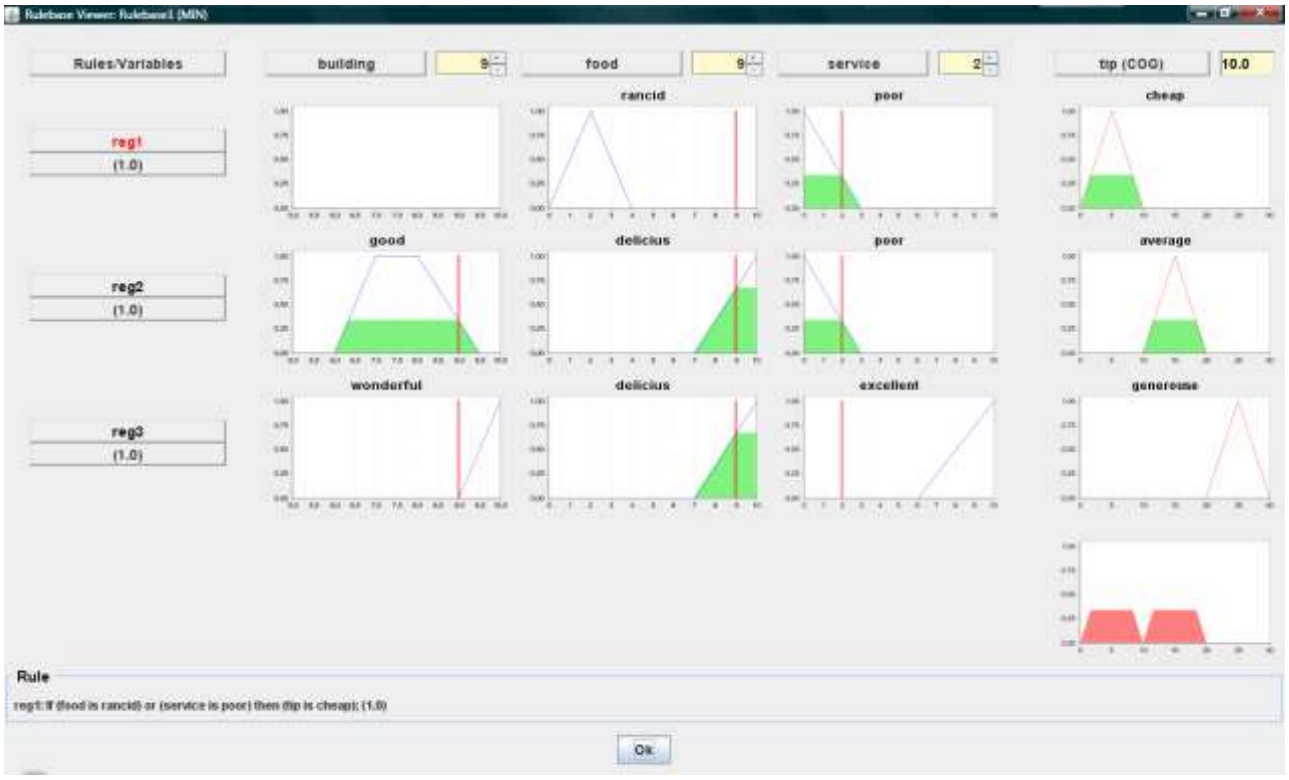
Rule Name: RULE1
Weight: 1
Connector: and
Operator: MIN

Antecedent — Variable: food; Modifier: none; Term: rancid
Consequent — Variable: tip; Term: generouse

Fig. 13: TskRulebase Editor

Systems Verification

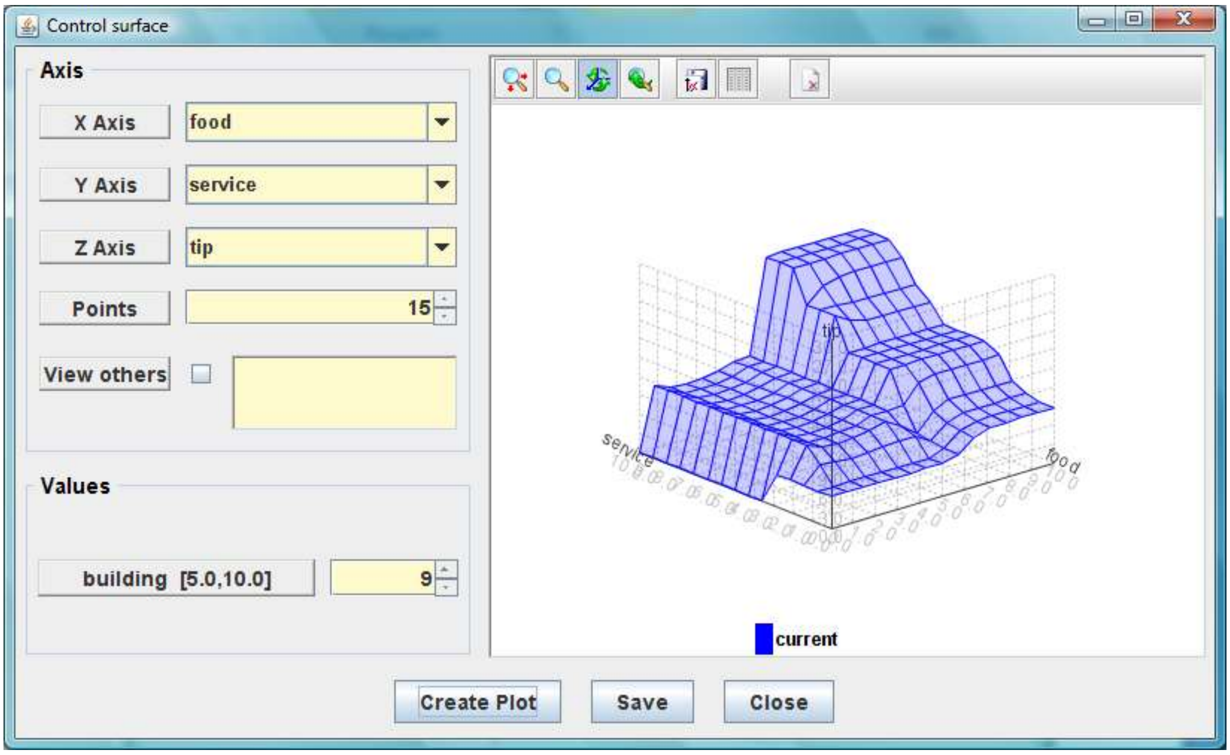
The verification stage in the fuzzy system design process consists in studying the behavior of the fuzzy system under development. The aim of this stage is the detection of probable deviations on the expected behavior and the identification of the sources of these deviations.
VisualFMLTool 0.1 environment covers the verification stage with two functionalities. The first one is "*Run*" that shows results of fuzzy inference process. The second one is "Surface 3D Plot" that implements a three-dimensional graphical representation of the system behavior.
The aim of the *"Run"* functionality is to monitor the fuzzy inference process in the system, to show graphically the values of the different output variables for a given set of input values.



reg1: if (food is rancid) or (service is poor) then (tip is cheap); (1.0)

Fig. 14: Rulebase viewer

The previous figure shows the "*Rulebase viewer*" window. The window title contains the name of rule base and the activation method. Each rule is a row of plots, and each column is a variable. Each plot has a name that represents the terms of variables that build the rule. Notice that when there is a plot which is blank, this corresponds to the characterization of none of terms for that variable in that rule. The rule names are displayed on the left of each row. The real number under rule name represents the weight of rule. You can click on a rule name to view the rule on the bottom, whereas the tooltip, shown when mouse is on the rule name, informs about connector method of rule. The fourth plot in the column of plots associated with an output variable represents the aggregate weighted decision for the given inference system. This decision will depend on the input values for the system. The initial input value for an input variable is the minimum value of its universe of discourse. The variables and their current values are displayed on top of the columns. For input variables there is a spinner to insert input value to fuzzy inference process, whereas for output ones there is a not editable field to display the value that results from process. With name of output variable is shown also the defuzzification method of variable, whereas the tooltip, shown when mouse is on name of output variables, informs also about accumulation method.


Control surface

Axis

X Axis: food

Y Axis: service

Z Axis: tip

Points: 15

View others

Values

building [5.0,10.0]: 9

Fig. 15: Control surface

The previous figure shows the "*Surface Viewer*". The graphical 3-dimensional representation illustrates the behavior of a fuzzy system, a surface plot showing an output variable as a function of two input variables. Hence, the system to be represented must contain at least two input variables and an output one. If the system contains more than two input variables, the not selected ones have to be specified by the user. The window is divided into two parts: the left one is dedicated to configuring the graphical representation, while the right part of the window is occupied by the surface plot. The configuration zone is formed by three pulldown lists which allow the selection of the variables assigned to each axis and a set of fields dedicated to introducing the fixed values of the not selected input variables. The field, named "*Points*", contains the number of points used in the partition of the X and Y axis. This is an important parameter because it determines the representation resolution. A low value in this parameter can exclude important details of the system behavior. On the other hand, a high value will make it difficult to understand the represented surface, as it will use a very dense grid. The default value of this parameter is 10, the minimum one is 5 and maximum one is 100. The bottom of the window shows a command bar with three options: "Create Plot", to actualize the graphical representation with the present configuration, "Close", to exit, and "Save", to save the current surface in order to compare it with a new surface obtained from a different set of values given to not selected variables or from modifications to the rule base. However, the comparison is possible between surface created using the same input variables on X and Y axis and output one on Z axis.

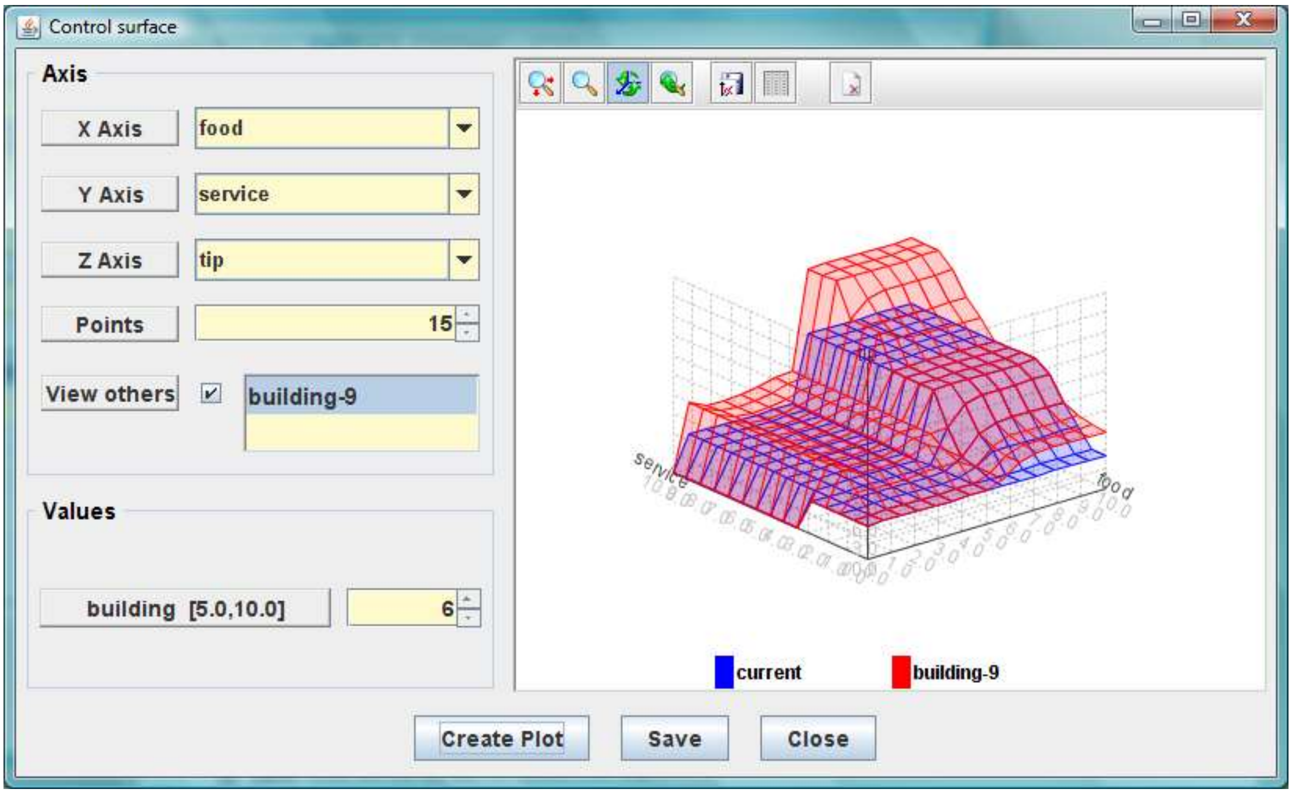
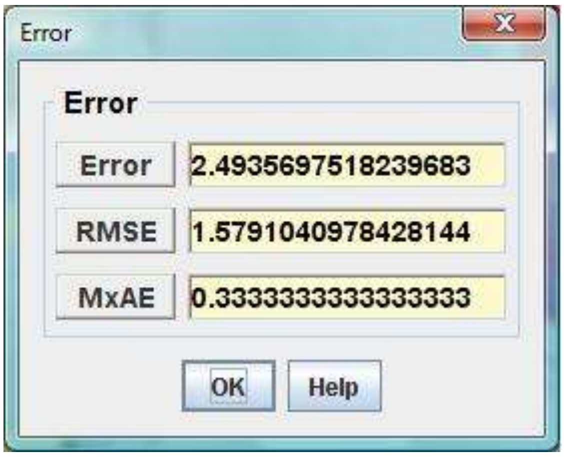


Fig. 16: Comparison of surfaces

In the previous figure it is shown a comparison between a current surface built with the variable building set to 6, and the surface, named "building-9" when it was saved, built with the variable building set to 9.
By means of toolbar shown at the top of graphical representation it is possible rotating the surface, zooming, saving it as a png file, and showing all the points of surface. When two surfaces are compared and they have got the same number of points, you can calculate error between them, too.


Error

Error

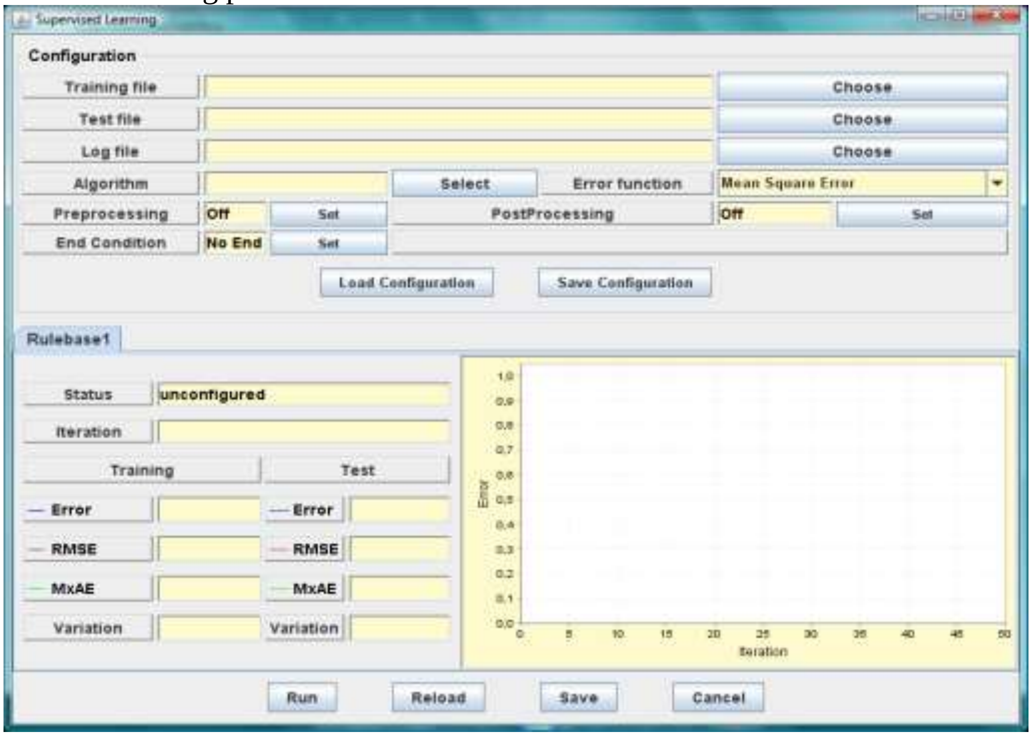
Error	2.4935697518239683
RMSE	1.5791040978428144
MxAE	0.3333333333333333

Fig. 17: Error calculation

The previous window shows the calculated error. The field with label "Error" contains the error calculated by means of MSE (Mean Squared error) function. Instead RMSE value indicates the square root of MSE and finally MxAE value indicates the max absolute value of error.

Systems Tuning (under development)

Although the usual approach to design a simple fuzzy system is to translate the knowledge of a human expert expressed linguistically, problems can appear because the way to implement the translation is not unique and/or the knowledge is not always available or possible to realize. This I why many researches have worked and are still working on applying automatic tuning techniques to fuzzy systems. If the tuning is applied only to the membership functions representing the antecedents and consequents of the rules, the fuzzy system is usually called a *self-tuning* system. In a more general case, if the rule base Isi also tuned, the system is usually called a *self-organizing* fuzzy system. The tuning techniques that have been used to adjust a fuzzy system can be grouped into the following categories: (a) meta-level heuristic rules, which implement the knowledge of an expert on tuning the system; (b) supervised and non-supervised algorithms taken from the neural network domain, which require a set of numerical training data; (c) reinforcement learning, which is applied when the only feedback about the system performance is a reward/punishment signal; and (d) genetic algorithms, which try to improve the system performance according to a set of objectives included within adequate fitness function. When input/output training data are available, supervised learning algorithms usually provide the best results for tuning a fuzzy system. For this reason they are the algorithms we have considered to implement the functionality “Tuning” of Visual FML Tool. The fig. 18 shows the window “Supervised Learning” which is used to configure and to execute the learning process.



Error function: Mean Square Error
Preprocessing: Off
PostProcessing: Off
End Condition: No End
Status: unconfigured

Fig. 18: Window Supervised Learning

The window is divided into two parts: at the top the area is dedicated to insert data to configure the learning process, whereas at the bottom the area is used to execute it. The first step in the configuration is to select a training file which contains the input/output data of the desired behavior. Besides you can select also a test file, written as the training file, used to check the generalization of the learning, and a log file to save the learning evolution in an external file. The following steps are selecting the learning algorithm and the error function (which measures the deviation of the fuzzy system behavior from the desired input/output pattern set). The tool makes available the following algorithms: BackPropagation[14] e Conjugate Gradient[21] belonging to gradient descent algorithms. However, the gradient of the error function cannot always be calculated because it can be too costly or not defined. In this case, optimization algorithms without derivatives can be employed. Regarding this category of algorithms, the tool offer Powell's method. This kind of algorithms is much slower than the previous ones. At the end, the tool makes available a statistical algorithm: Blind Search which may provide good results when the number of parameters is low. Regarding the error functions, the tool offer only two ones: considering N as the number of data patterns, M as the number of output variables in the system, y_{ij} as the j^{th} output generated by system for the i^{th} pattern, $\tilde{y}_{ij}$ as the correct output value written on training file, and r_j as range of the j^{th} output variable, we have:

- Mean square error (MSE):

$$MSE = \frac{1}{N} \cdot \frac{1}{M} \cdot \sum_{i,j} \left(\frac{y_{ij} - \tilde{y}_{ij}}{r_j}\right)^2, i = 1, \ldots\ldots; j = 1, \ldots M$$

- Mean absolute error (MAE):

$$MAE = \sum_{i,j} \left(\frac{|y_{ij} - \tilde{y}_{ij}|}{r_j}\right), i = 1, \ldots\ldots; j = 1, \ldots M$$

An important feature of learning process offered by tool is to simplify the tuning process before (preprocessing) or after (postprocessing) it. One of the simplifications consists in detecting and deleting those rules that are never activated sufficiently by any of the training input/output patterns. The last configuration step is to select the end to select the end condition to finish the learning process. This condition can be a limit imposed over the number of iterations, the maximum error goal, or the maximum absolute or relative deviation (considering either the training or the test error).

Systems Synthesis (under development)

The main aim of system synthesis is to generate a system representation that could be computed on different hardware and software environments. In details, this step enables to translate an FML representation of a controller into two different types of implementation models: software representations and hardware representations. The software synthesis generates a system representation in a high level programming language. The hardware synthesis generates a microelectronic circuit that implements the inference process described by the fuzzy system. Currently, VisualFMLTool provides only software synthesis and only for Java language. The synthesis is implemented thanks to Extensible Stylesheet Language Transformations (XSLTs) modules. Indeed, XSLT is able to convert the FML fuzzy controller in a general purpose computer language using an XSL file containing the translation description. So, VisualFMLTool executes synthesis in Java language thanks to a defined XSL file. Therefore, when you click on "*To Java*" from menu "*Synthesis*", a simple dialog window opens to select a directory in which saving Java classes generated from tool in automatic way by means the file xslt.