A FRAMEWORK FOR ADVANCED FUZZY LOGIC INFERENCE SYSTEMS

J.M. Perronne, C. Petitjean, L. Thiry, M. Hassenforder

ESSAIM
12, rue des frères Lumière
F-68093 Mulhouse Cedex
{ JM.Perronne, C.Petitjean, L.Thiry, M.Hassenforder }@uha.fr

Abstract: This paper describes how high-level of Object-Oriented concepts can be used to provide a generic, portable and polymorphic Fuzzy Logic framework. It highlights the way in which such OO concepts allow the extension of the programming language idioms with the semantics of the fuzzy logic field. A progressive approach presents; in a first step Fuzzy Logic systems; then, relevant classes, design patterns and architectures are identified. The considered aspects cover the composite structure, the polymorphic behaviour and the building of a system of fuzzy expressions. Finally, an example illustrates how the framework can be used in a conventional design strategy. Copyright © 2001 IFAC.

Keywords: Fuzzy systems, Object modelling techniques, Software engineering, Computer-aided system design

1. INTRODUCTION

The aim of this paper is to design portable fuzzy logic inference systems. It will be expressed using the syntax of the underliyng programming language with the fuzzy semantics. The portability of the fuzzy logic inference systems will be assumed by the programming language. Moving the designed system from the design computer to the final target (in-line), just needs a compiling stage with the final target compiler. So, the expertise domain of the Fuzzy Logic (variables, membership functions, operators, rules, ...) field have to be captured and have to take the form of an extension of idioms that the user has to manipulate. Moreover, the fuzzy system has to be flexible and have to provide polymorphic operators. They allow the studying of the impact of any kind of operators without redesining tasks.

When a fuzzy inference system has to be designed, the only target of the designer is to set up the right fuzzy subsystem in order to master his problem. The task of writing software in this scope must not be an obstacle in the design process. It would be convenient if the designer had tools that took charge of the software complexity and responded to his problems. To create such tools, Object Oriented paradigms - among which Design patterns and software frameworks - can be exploited as valid modelling and implementation techniques in advanced control engineering (Maffezoni 1999). Design patterns are a generic solution to recurring problems which makes it easier to reuse successful designs and architectures. Software frameworks are semi-complete applications, which can be defined as reusable designs for an entire application or for part of an application (Fayad, 1999); they automate the generation of classes of an application. A Fuzzy Logic Framework involving several Design Patterns is presented here; it allows the building of Fuzzy Logic applications using the sole specifications required by a specific Fuzzy Logic system. UML diagrams (Muller, 1997) will be used to describe the study of the framework and design patterns.

2. PROBLEM ANALYSIS

A general fuzzy system must be analysed; pertinent objects and classes of the right granularity must be found and described with the appropriate vocabulary (Booch 94). From these considerations, a framework, which uses the elements discovered must be developed in order to make the designer's task easier.

Fuzzy Logic allows mapping from a given input to an output. This is illustrated by the following general fuzzy system:

If x is "low" and y is "medium" then w is "low"
If x is "low" or y is "high" then w is "high"

This involves components such as variables, membership functions, Fuzzy Logic operators (and, or ...) and "if-then" rules (Zadeh 73). There are two types of fuzzy inference systems, the Mamdani and the Sugeno types; their description can be found in the references (Mamdani 75) and (Sugeno 85). The major differences between these two methods concern the implication part of a fuzzy rule and the way in which a fuzzy inference system is defuzzified.

The building of any type of Fuzzy Logic inference system needs the use of classical or particular operators and membership functions suited to a specific system. In the case of particular operators and membership functions the user must be able to define them easily.

At present, any kind of static Fuzzy Logic inference systems can be developed. However, if different types of fuzzy operators must be tested to determine the best structure of the fuzzy system, a user usually must rebuild a new fuzzy system. This process is time and money consuming and does not lead to an optimised design process. So, in an advanced fuzzy framework, a user must be able to change the nature of the operators without creating a new system at any time.

To summerise what a designer has to do – i.e. to design and use a fuzzy subsystem - the different stages involved in such a process have to be identified. The use-case and sequence diagrams presented in figure 1 show, successively, the designer's requirements and interactions between the designer and the framework during design and use processes.

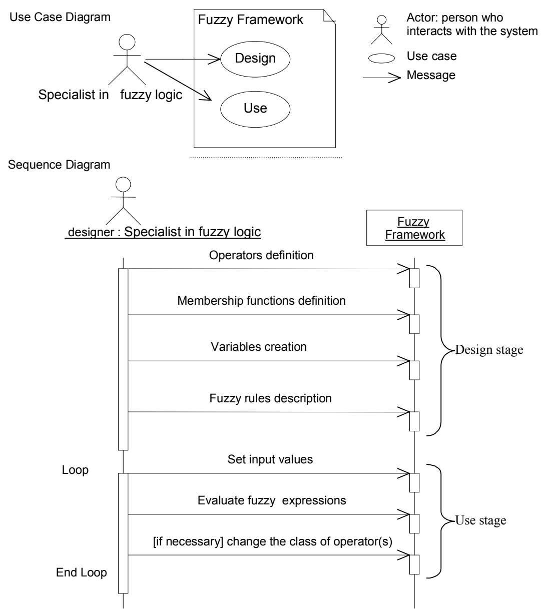

Use Case Diagram
Fuzzy Framework
Actor: person who interacts with the system
Design
Use case
Message
Specialist in fuzzy logic
Use
Sequence Diagram
designer : Specialist in fuzzy logic
Fuzzy Framework
Operators definition
Membership functions definition
Variables creation
Design stage
Fuzzy rules description
Set input values
Loop
Evaluate fuzzy expressions
Use stage
[if necessary] change the class of operator(s)
End Loop

Fig. 1 Use-case and sequence diagram of the building and use of a Fuzzy Logic inference system

3. SIMPLE FUZZY EXPRESSION ARCHITECTURE

Fuzzy rules suggest that a Fuzzy Logic system can be expressed as a collection of expressions. So, fuzzy operators and values look like operators and values in arithmetic expressions. In the above fuzzy rule system example, the different parts of the expressions are:

- Variables: x, y, w.
- Membership functions: "low", "medium", "high".
- Fuzzy Operators: and (T-norm operator), or (T-conorm operator), then (implication operator), aggregation.

3.1 Composite facet of the architecture

In fact, each component (variables, membership functions, operators) can be considered as an expression which can be evaluated. This implies that there is a hierarchy of expressions, which leads to a composite structure. The behaviour of this structure will be obtained by the evaluation of the composite expressions.

The architecture of a fuzzy system distributes the different components into a hierarchical tree structure. Figure 2 illustrates the composite hierarchical structure of the fuzzy example given above. Moreover, each element has to be treated uniformly, whatever its nature; each element is an expression which must be evaluated. The *Composite* Design Pattern allows such construction and such behaviour [Gamma, 95]. Each node of this tree must have a unified behaviour, which takes the form of an evaluation. The recursive evaluation of the tree gives the evaluation of the fuzzy system. Another Design Pattern called *Interpret* [Gamma, 95] illustrates this type of solution; in fact, this pattern also involves the composite one. Each class of components involved in such a tree describes a specific type of expression (binary expression, unary expression, value…) and how to interpret or evaluate it. This kind of tree can also be seen as the abstract syntax tree of the fuzzy language.

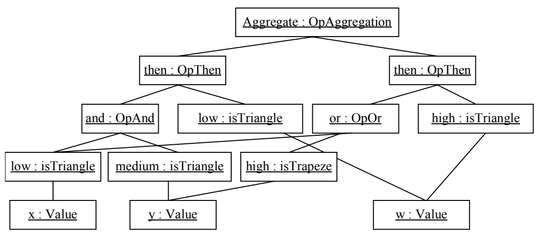


Fig. 2 Composite structure of the example

Now, the structure of fuzzy expressions can be defined using involved Design Patterns. The result leads to the architecture depicted in figure 3. The different mechanisms involved are depicted, in the context of fuzzy expressions, as follows.

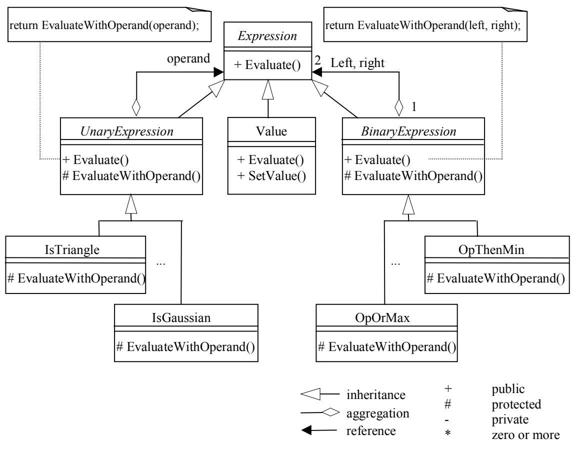


Fig. 3 Class diagram of a composite fuzzy expression architecture

3.2 Implementation of Composite and interpret design patterns

An abstract class Expression declares the interface for expressions for the interpreting behaviour. The abstract Evaluate operation that is common to all nodes in the abstract syntax tree (composite structure) is defined.

A terminal expression illustrated in the present case by the *Value* class implements the interpret operation associated with the terminal symbol which is a fuzzy value. The evaluation of this type of expression gives an arithmetic value.

Two abstract non-terminal expressions depicted by the *UnaryExpression and BinaryExpression* classes define behaviours for components with children. They store operands and implement the *Evaluate* operation for fuzzy binary expressions (and, or, then, aggregation operator…) and fuzzy unary expressions (not operator, membership functions).

The *UnaryExpression* class is composed of an operand named *operand*, which is an expression.

The *BinaryExpression* class is composed of two operands named *left* and *right*, which are expressions.

The evaluation of instances of these classes induces the call of the abstract *EvaluateWithOperand* operation and provides it with the operand(s). Each subclass of these non-terminal expressions have to define the concrete behaviour in order to provide concrete operators or membership functions. This mechanism also represents a Design pattern called template method [Gamma, 95].

3.3 Template method design pattern

This pattern allows a common use of different kinds of classes by delegating specific parts of the behaviour to subclasses. Here, the *UnaryExpression* and *BinaryExpression* classes are evaluated by the call to common *Evaluate* operation, but the behaviour is given by the specific *EvaluateWithOPerand* operation. The *Evaluate* operation provides the invariant part of the behaviour; it consists of the operands transfer. The major advantage of this mechanism is the segregation between the point of view of the user, who wishes to evaluate a fuzzy expression and that of the designer, who has to define the concrete behaviour of an operator.

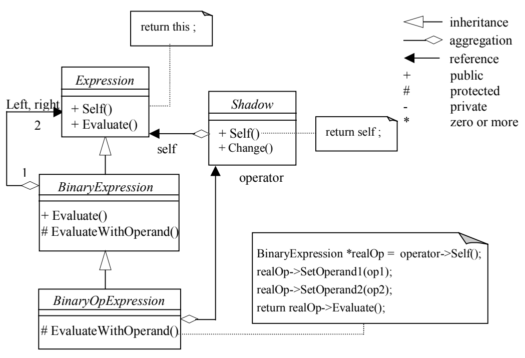
With this architecture, it is possible to build simple fuzzy expressions.

4. ENHANCED FUZZY EXPRESSION ARCHITECTURE

Moreover, a polymorphic behaviour can also be imagined in order to test the effect of different kinds of fuzzy operators without changing the structure of a system.

4.1 Polymorphic Expressions

To support this extension, composite classes called *UnaryOpExpression* and *BinaryOpExpression* are added so as to allow the handling of composite expressions as objects. These classes are expressions and inherit from the *UnaryExpression* and *BinaryExpression* classes; thus they have operands and can be evaluated. To complete these classes, an operator class member is added; so, an expression such as «*x is "low" and y is "medium"*» can be instantiated as an independent object where the expressions «*x is "low"*» and «*y is "medium"*» are the operands and the «*and*» expression is the operator. To obtain a polymorphic evaluation of this object the behaviour of the operator «*and*» has to be changed without rebuilding it. In this case, the operator has to appear as it changes its class. A mechanism that allows the behaviour of an object to be altered when its internal state changes must be set-up. This mechanism used for fuzzy expressions is depicted in figure 4; only the *BinaryExpression* class is shown, but the *UnaryExpression* class *follows* the same pattern.



```
BinaryExpression *realOp = operator->Self();
realOp->SetOperand1(op1);
realOp->SetOperand2(op2);
return realOp->Evaluate();
```

Fig. 4 Expression and polymorphism

So far, there are two kinds of expressions:

- Simple expressions (*BinaryExpression*, *UnaryExpression)*, where the evaluation is direct. They will be used as super-classes for fuzzy operators and membership functions.

- Enhanced expressions (*BinaryOpExpression*, *UnaryOpExpression)* where the evaluation needs an external operator. They will be used to build fuzzy expressions such as «*x is "low" and y is "medium"*», where the and operator can be changed at any time without rebuilding this expression.

To determine the polymorphic behaviour, the enhanced expressions do not use a real operator, but a *Shadow* operator, which provides and holds a link to a real operator. To perform the polymorphic evaluation, the shadow operator must be reconnected to another kind of real operator, and so, the behaviour of the expression changes but the structure of the expression remains constant. Overriding the *EvaluateWithOperand* method of the *BinaryOpExpression* specifies the evaluation of an enhanced expression. The evaluation process includes three stages (figure 4):

- At run-time, the real operator is found by request, using the *Self* method of the shadow operator.

- The operands are transferred to the real operator.

- The real operator evaluates the expression.

Figure 5 shows, before evaluation, the object composition of enhanced fuzzy expressions as :

x is "low" and y is "medium"
a is "low" and b is "medium".

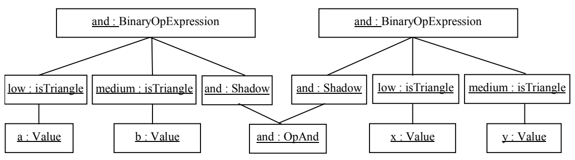


Fig. 5 Composite structure of an enhanced fuzzy expression.

With such a mechanism, an operator can also be used by several enhanced expressions in the same Fuzzy Logic inference system, because the real operator and its operands are only determined at run time according to the context of the expression.

4.2 *Factoring enhanced expressions*

Enhanced fuzzy expressions improve the skills of this fuzzy architecture but they are not easy to manipulate. Indeed, the desired behaviour must be obtained by putting together their different parts (arguments, shadow operator, real operator) and holding a reference on the shadow operator in order to reconnect it to real operators. The problem can be solved by using a factory that will create and manage the enhanced fuzzy expressions and their components; figure 6 illustrates such a structure. In this diagram, two classes of factories are depicted.

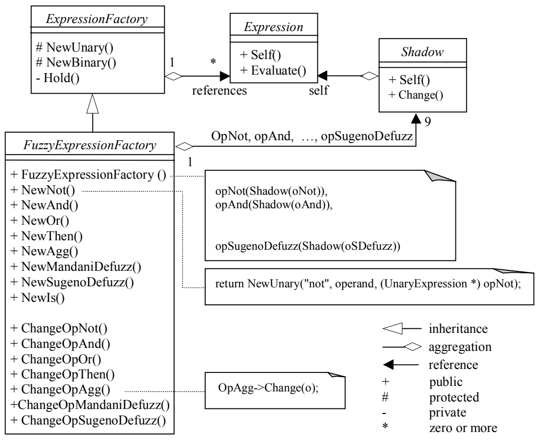


Fig. 6 Factory and the building of enhanced expressions

The *ExpressionFactory* class proposes:

- The basic mechanisms to build expressions using two methods (*NewUnary*, *NewBinary*). The assembly and the naming processes are carried out.

- A basic mechanism to manage the memory. All the expressions created by the factory are referenced and will be destroyed when the factory disappears, so a really simple expression garbage collector is set up.

The *FuzzyExpressionFactory* class inherits from the *ExpressionFactory* class and takes account of the management of the components involved in the polymorphic behaviour described above. This class proposes:

- The mechanism which builds shadow operators and binds them with real operators using a value constructor where shadow operators are instantiated and linked to real operators provided as arguments.

- Different methods which construct fuzzy expressions with fuzzy semantics. The user of this factory merely has to request the desired fuzzy expression to obtain it. For example, to obtain a fuzzy expression *and*, the user has to use the *NewAnd* method and to provide the right operands. The method puts together the different parts of the enhanced expression and determines the right shadow operators to be used.

- Methods for changing the behaviour of an operator. The *ChangeOp…* methods allow shadow operators to be reconnected to other real operators. So, the user can change the behaviour of a fuzzy system to test the influence of a particular fuzzy operator on it without modifying its structure. The user only has to invoke the matched *ChangeOp…* method of the factory to do so.

With a class like *FuzzyExpressionfactory* the enhanced expressions can be created and manipulated as simply as basic expressions and a polymorphic behaviour can be managed in a really simple manner in every composite system of expressions.

5. ILLUSTRATED EXAMPLE

A basic Fuzzy Logic system is used to illustrate the nature of Fuzzy Logic inference systems. This example deals with the characterisation of a driving situation. The inputs are the speed of a passenger car and the experience of the driver. The output is the danger quotation of a situation, which shows how dangerous the situation is. The speed can be low, medium or high; the driver can be described as a novice, a regular or experienced driver and the situation can be normal, critical or dangerous. Figure 7 illustrates several components of this fuzzy system.

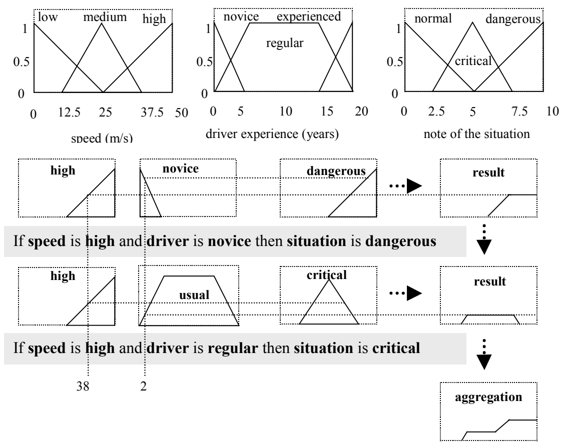


Fig. 7 Example of a Fuzzy Logic system

This example shows that this set of mechanisms forms a framework and allows the building of any type of Fuzzy Logic inference system. All that the user has to do is to specialise some classes as special fuzzy operators or membership functions to match the framework to a particular case. The example considers the use of this framework in a Object Oriented Programming Language context (C++/Java); the framework acts so as to extend the idioms of the language (Coplien 1992). So, reusable expressions with fuzzy semantics and their composition rules are added to the language.

In the next listing, all the stages of the design process of a Fuzzy Logic inference system are performed, as depicted in the sequence diagram of figure 1.

```
//Design stage

// operators definition
NotMinus1 opNot;
AndMin opAndMin;
AndMult opAndMult;
OrMax opOr;
ThenMin opThen;
CogDefuzz opDefuzz;

//fuzzy expression factory
FuzzyExpressionFactory factory (
&opNot,&opAndMin,&opOr,&opThen,&opOr,&opDefuzz
);

//membership functions definition
IsTriangle high(25, 50, 75);
IsTriangle novice(-5, 0, 5);
IsTrapeze regular(0, 5, 15, 20);
IsTriangle critical(2.5, 5, 7.5);
IsTriangle dangerous(5, 10, 15);

// variables definition
Value speed, driver, situation(0, 10, 0.1);

Expression *e=
    factory.NewAgg(
        factory.NewThen(
            factory.NewAnd(
                factory.NewIs(&speed,&high),
                factory f.NewIs(&driver,&novice)
            ),
            factory.NewIs(&situation,dangerous)
        ),
        factory.NewThen(
            factory.NewAnd(
                factory.NewIs(&speed,&high),
                factory.NewIs(&driver,&regular)
            ),
            factory.NewIs(&situation,critical)
        )
    );

//defuzzification definition
Expression *system =
    factory.NewMamdaniDefuzz(&situation,e);

//Use stage

//apply inputs
float sv,de;
cout << "enter speed value and driver experience ";
speed.SetValue(sv);
driver.SetValue(de);
//evaluation
cout << "result situation : " << system->Evaluate();

//change a and operator : AndMult replaces AndMin
factory.ChangeOpAnd(&opAndMult);
//new evaluation with the same system
//but with an other operator and
cout << "result situation : " << system->Evaluate();
```

This example assumes that the concrete classes of fuzzy operators, such as *AndMin*, *ThenMin* based on the minimum of operands and the concrete classes of *IsTriangle* membership functions based on a triangular shape have been implemented. The concrete defuzzification operator named *MamdaniOpDefuzz* has also been implemented and performs a Mamdani defuzzification process through the computation of the centre of gravity of a resulting shape.

The design part of the listing presents the building of the example (figure 7). The use part shows a first evaluation using the defined system, followed by a second evaluation where the initial AndMin operator is replaced by an AndMult operator.

6. CONCLUSION

This study has shown how object oriented concepts, design patterns and frameworks give solutions for the building of systems, which responds to a class of problems. Using the framework proposed, any kind of fuzzy system can easily be set-up. The expertise domain of the Fuzzy Logic field is captured in the framework and allows the framework to be reused by any user, which saves time and money. Thus, the design and maintenance processes are improved and facilitated. As the example shows, reuse does not mean using class libraries, but takes the form of an extension of idioms that the user has to manipulate. Thereby, the user expresses himself in his domain of expertise (Fuzzy Logic field) with the power of the programming language (in this case c++), which supports the framework. This framework was used in a project dealing with a fuzzy observer for fault detection (Amann 1999) based on fuzzy rules estimators. To extend the principle, template expressions were used, as they can manipulate any type of values. In this context, fuzzy estimators had to handle expressions which deal with vectors. The instantiation of the generic expressions specialised with vectors allowed the immediate reuse of the framework.

7. REFERENCES

Amann, P., Perronne J.M., Gissinger G.L., Frank P.M (1999). Identification Of Fuzzy Relational Models for Fault Detection. 14 th World Congress of IFAC Beijing 99, Vol K, pp309-314.

Booch G. (1994). Object-Oriented Analysis and design with applications, 2nd edition, Addison-Wesley.

Fayad M.E. et al. (1999). Building application frameworks: object-oriented foundation of framework design, Wiley and Sons Inc

Coplien J.O (1992). Advanced C++ programming styles and idioms, reading Massachussets. Addison-Wesley.

Gamma, E.Gamma et al (1995). Design patterns, elements of reusable O.O. software. Addison-Wesley

Maffezzoni C. et al. (1999). Object-Oriented models for advanced automation engineering. Control Engineering Practice 7, pp 957-968.

Mamdani, EH and S. Assilian (1975). An experiment in linguistic synthesis with Fuzzy Logic controller. International Journal of Man-Machine Studies, Vol 7, No 1, 1-13.

Muller, P.A. (1997). Instant UML, Wrox Press.

Sugeno, M. (1985). Industrial application of fuzzy control. Elsevier Science Pub. Co.

Zadeh, L.A (1973), Outline of new approach to the analysis of complex systems and decision processes, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 3, No. 1, 28-44