XFL: A LANGUAGE FOR THE DEFINITION OF FUZZY SYSTEMS

D. López, F. J. Moreno, A. Barriga, S. Sánchez Solano

Instituto de Microelectrónica de Sevilla - Centro Nacional de Microelectrónica
Avda. Reina Mercedes s/n, (Edif. CICA)
E-41012, Sevilla, Spain

Sixth IEEE International Conference on Fuzzy Systems (FUZZ-IEEE’97),
Abstract

This paper presents the main characteristics of XFL, a language for fuzzy logic based systems. Its more relevant features are the ability to define potentially complex systems and the independence from the particular techniques employed for implementing fuzzy operations. This language has been used as the base for several hardware- and software-oriented development tools, which are also briefly introduced.

1. Introduction

Fuzzy logic has been successfully applied in the recent years to fields such as decision-making systems, control, image recognition and non-linear systems modelling. Fuzzy logic definitions are able to capture, by means of a “natural” formalism, the knowledge that humans express through language. This allows a reduction of the complexity in the mechanisms used to perform a specific task from its linguistic description.

Most of the tools currently available for the development of fuzzy logic based systems are based in a limited set of fuzzy operations. By fuzzy operations we mean those functions which define the information processing performed by the system: implication functions, fuzzy connectives, defuzzification methods, etc. This implies that the definition of a fuzzy system using these tools is somehow bound to the operations implemented by the tool in use. Three kinds of drawbacks derive from this situation. First, there are limitations in the comparability of similar systems designed with different tools. Second, implementation-oriented details can hide the original structure of the system under development. Third, this situation makes more difficult the comparison among different design alternatives for each fuzzy operation (making necessary the use of ad-hoc methods [1]) and the evaluation of new formalisms.

A common criticism made to fuzzy logic is that it has not been applied to real-world complex systems. In spite of the fact that there exist several examples of such systems in the literature, like [2] and [3], it is true that current development tools offer few facilities for working with high complexity systems. For example, these tools do not provide the hierarchical aggregation of rulebases [4], or do not allow that rule antecedents contain expressions of arbitrary structure and complexity.

When we started the work on a new version of our fuzzy development tool, Xfuzzy [5], one of our first goals was to establish a specification method that overcomes the above mentioned problems. This lead us to define the following objectives:

- The specification method should be independent from the specific operations used in each case. Furthermore, the use of any of these operations (or combination of them) should be configurable in the tools based on this method.
- The method should not impose any constraint with respect to the complexity of the system, either in the structure of the rulebases or the definition of the rules themselves.
- The method should demonstrate its validity by allowing the development of a set of flexible tools, able to produce open implementations, i.e., these implementations should be easily integrated with other modules (created by any other means and not necessarily based on fuzzy logic) to build a complete system.

This paper presents the results of this effort: the XFL language and the tools based on it. The next section describes the main characteristics of XFL and the basic blocks of a specification that uses it. The details about the definition of linguistic variables and rulebases are discussed in the following section. The mechanisms for the definition of fuzzy operations and the use of these definitions by XFL-based tools are introduced in section 4. Finally, section 5 describes the XFL-based tools currently developed and section 6 presents some conclusions.

2. The XFL language

XFL is a language for the definition of fuzzy logic based systems designed to ease the description, verification and synthesis of fuzzy elements, according to the objectives
described in the above section. The definition of a fuzzy system by means of XFL is made of two basic parts: type declarations and the specification of the module(s) that make up the system. The first part contains the definition of the variables the system is going to work with, including their universes of discourse and the membership functions corresponding to each of the fuzzy sets defined for these variables. The second defines the structure and contents of the rulebase controlling the behavior of the system.

The model underlying a specification written in XFL consists of three phases [6]:

- Input variables are fuzzified, assigning for each of them a value to the applicable membership functions.
- The rules in the rulebase are applied, using the specified connectives and implication functions. This process yields a global membership function for each of the output variables of the system.
- Output variables are defuzzified using the results of the previous phase, thus obtaining the output values for the system.

XFL supports a set of predefined membership function classes that includes most of the functions described in the literature [6],[7] and provides a method for defining generic functions as piecewise linear approximations.

As previously stated, XFL does not impose any restriction on the complexity of the rulebases controlling the behavior of the systems described by using it. This is true both for the structure of any individual rule and for the relationships among different rulebases inside the system.

An individual rule is an arbitrary combination of predicates involving the membership functions of its antecedents, using the connectives and, or and not. These predicates can be grouped by means of parenthesis and the traditional rules of precedence, associativity and distributivity are applied, thus guaranteeing a “natural” interpretation of rules.

XFL supports two different modes for calculating the output membership functions. In the first mode, each rule contributes to the output membership function independently, i.e., the modus ponens derived from the implication function is applied once for each rule. This mode, which is the default when working with XFL, is the most commonly described in the literature and is typically applicable to control and decision-making applications. In the second, the results of all rules applicable to a particular linguistic label in the consequent are composed into a support value for the corresponding membership function, by means of the connective c. This support value is used as input to the implication function. This mode (which we call rule preaggregation) is specially applicable to classifier systems.

In any case, the value for the membership function for any of the output variables is obtained by composing the output values from the implication function by means of the connective also [6].

When defining a compound structure for system behavior, XFL supports two modes for the composition of rulebases. In the parallel mode a rulebase is simply split in several component rulebases (which can share common inputs) with independent outputs. It does not imply any relationship among the component rulebases. This mode of composition may be used for integrating several existing rulebases in a new one with more inputs and outputs, or for simplifying the format of the rulebase of a complex system. The serial mode assumes that at least one of the outputs of the first rulebase acts as input for the second. The shared outputs of the first rulebase determine the support value for each membership function corresponding to the shared inputs in the second rulebase, thus allowing the setup of hierarchical rulebases.

![Figure 1. Model for a hierarchical rulebase using XFL compositions](image)

Both modes of composition can be, as in the case of individual rules, arbitrarily combined to define the structure of the system rulebase. This way, for example, a rulebase composed using serial mode with other two rulebases that are composed in parallel corresponds to a model in which the primary rulebase controls rule activation in the secondary rulebases, which provide the system outputs, as shown in Figure 1.

Although a XFL definition is inherently independent of the defuzzification method used, the configuration facilities of the tools based in the language provide constructs specifically tailored for the two paradigms that support the basic structure of methods described in the literature [7],[8],[9]. The first consists in an exploration through the universe of discourse of the output variable. The values of each point and the value of the aggregated membership function are used. This paradigm is used by methods such as Center of Area, Mean of Maxima, etc. In the second the exploration is made through the linguistic universe of the variable, i.e., through the fuzzy sets defined for it. In this paradigm the activation grade of each rule (or membership function) and the parameters of the membership functions are accessed. If this defuzzification paradigm is used, it must be taken into
account that both the implication function and the connective also (and the preaggregation connective c, if it is in use) are not applicable, since the mechanisms for implication and aggregation are implicit in the defuzzification process. This paradigm is used by most of the so called simplified methods, such as Fuzzy Mean, Level Grading, Quality, etc.

3. The structure of an XFL specification

3.1. Type definition

The most relevant characteristic of the way in which universes of discourse and membership functions are defined (what is called type definition) in XFL is the support for inheritance mechanisms. These mechanisms are intended to simplify the specification of complex systems and to ease module reuse. A type definition in XFL has the following format:

\[
\text{type} \text{ Identifier BaseType \{ \\
\text{MembershipFunction} \\
\text{MembershipFunction} \\
... \}}
\]

Where BaseType contains a reference to the type (or types) from which the defined type derives. This reference can be done to a predefined type or to any type whose definition has been already made in the specification. XFL provides two predefined types: integer and real. When a type definition uses one of these predefined types, BaseType takes the form:

\[
<\text{integer} | \text{real}> \langle [\text{cardinality}] \rangle \langle \text{range} \rangle
\]

Where range are the values defining the universe of discourse for the variables of this type. It is specified by means of two numerical values with a < character as separators. The cardinality is an optional parameter that can be used to specify the number of distinct values that are going to be considered in the universe of discourse for a variable of this type. For integer types it defaults to the number of integers in the defined range, while for real variables a default value (currently, 256) is assumed when not explicitly specified. Cardinality is used in applications such as the synthesis of digital hardware (it defines the number of bits to be used for the variables) or the implementation of defuzzification methods when calculating the step size for exploring the universe of discourse.

If the type being defined is a derived type, BaseType may contain a list of previously defined type identifiers (separated by commas), with the only restriction that all of them must finally derive from the same predefined type. The range of the derived type is obtained from the intersection of its ancestor ranges. The cardinality is the lowest cardinality defined for the ancestor types. A derived type inherits automatically all the membership functions defined by its ancestors, with the following precedence rule for the cases in which one function appears in more than one parent type: the definition corresponding to the last type in the ancestor list is used.

The definition of a new membership function (or the overloading of a previously defined one for a derived type) corresponds to a construct of the form:

\[
\text{Identifier FunctionClass ( PointList )}
\]

Where PointList contains values for the points relevant for defining the function, according to the class specified by FunctionClass. These values have no other constraint than those required by class coherence (for example, the apex of a triangular function cannot correspond to the same value as the points defining its base) and that the function must take at least a non-zero value inside the universe of discourse. The supported membership function classes are illustrated in Table 1.

<table>
<thead>
<tr>
<th>FunctionClass</th>
<th>PointList</th>
</tr>
</thead>
<tbody>
<tr>
<td>triangle</td>
<td>basis, apex, basis</td>
</tr>
<tr>
<td>rectangle</td>
<td>basis, basis</td>
</tr>
<tr>
<td>trapezoid</td>
<td>major basis, minor basis, minor basis, major basis</td>
</tr>
<tr>
<td>bell, ( \left( \frac{x-a}{b} \right)^2 )</td>
<td>a, b</td>
</tr>
<tr>
<td>sigma, ( \frac{1}{1 + e^{(x-a)/b}} )</td>
<td>a, b</td>
</tr>
<tr>
<td>delta</td>
<td>value</td>
</tr>
<tr>
<td>points</td>
<td>point:value, point:value,...</td>
</tr>
</tbody>
</table>

3.2. Rulebase definition

The basic element in the definition of the rulebase controlling the behavior of a fuzzy system in XFL is called a module. Each module has a set of variables (for input and output) and a specification of its structure. In general terms, a module definition has the form:

\[
\text{Identifier ( VariableList ) ModuleStructure}
\]

Where the identifier allows further references to the module in some other (compound) module. XFL requires specifications to define a module called system, which
specifies the global behavior of the system and whose input/output variables are those of the whole fuzzy system. Each of the variables in a module are defined in VariableList by means of:

\[ \text{TypeIdentifier } <?|!> \text{ Identifier} \]

The identifier will be used in any reference to the variable inside the module, while TypeIdentifier defines the type of the variable and the character ? or ! identifies the variables as an input or an output, respectively. Type and module identifiers are the only global symbols in a XFL definition. All other identifiers are local to the type or module in which they are defined.

As previously stated, XFL allows the definition of rulebases of an arbitrary (complex) structure. This feature is supported by the specification of module structures. There are two different ways for this specification: a set of rules can be defined, or the module can be specified in terms of a set of composed modules. For a module whose structure is defined in terms of a set of rules ModuleStructure is:

\[
\text{rulebase} \{ \\
\text{Rule} \\
\text{Rule} \\
... \}
\]

While for a module whose structure is refined in several interconnected modules, ModuleStructure takes the form:

\[
\text{components} \{ \text{ModuleReferences} \}
\]

Each individual rule has the following structure:

\[
\text{if Antecedent } \rightarrow \text{ Consequent}
\]

The antecedent of a rule is an arbitrary combination of expressions binding any of the input variables to any of the membership functions defined for its type. Each of these expressions is of the form:

\[
\text{VariableIdentifier is } \text{ MembershipFunctionIdentifier}
\]

The operators combining the expressions in the rule antecedent are identified by the character ~ for the not operator, the character & for the and operator and the character | for the or operator. The usual rules for associativity, distributivity and precedence hold. As usual, expressions can be grouped by means of parenthesis. The consequent of a rule is made of a list (comma separated) of expressions binding any of the output variables to any of the membership functions defined for its type.

When the structure of a module is defined in terms of the composition of a set of other modules, this definition consists in an arbitrary combination of references to the components modules connected by means of the parallel (symbol \&) and serial (symbol ;) composition operators. A reference to a module is made by including its identifier and instantiating its input/output variables to input/output variables in the parent module or to dummy interconnection variables in the case of serial composition.

The semantics for module composition follows the following rules:

- Modules composed using parallel mode does not share any constraint and model the decomposition of the parent module in simpler elements. Two parallel composed modules can share input variables (which are processed independently by each module) but never output variables.
- Modules composed using serial mode share at least one variable: output from the first and input to the second. In this context we can refer to dummy interconnection variables linking the two modules, since it is not possible to use variables instantiated from the parent module for communicating serially composed modules. Dummy interconnection variables act as internal data paths. They do not correspond with actual inputs or outputs and are used as the connection mechanism for this kind of composition.
- The association with the instantiated parent module variables and the link between dummy variables is made implicitly by name using parent’s module namespace. The obvious rules enforcing compatibility in types and modes (input/output) are applied. Input variables to the parent modules can be freely used as input variables to any number of component modules, while each output variable from the parent module can be only instantiated by one component module.
- All the variables in a component module must be associated either to a parent module variable or by means of a dummy linking in serial mode composition.
- The component module structure is arbitrary, taking into account that: a) Serial composition takes higher precedence than parallel; b) Serial composition is distributive with respect to parallel; and c) Both operators are associative. Serial composition is not, obviously, commutative.

Module composition can be grouped by means of parenthesis. Figure 2 shows the structure in terms of module compositions for the hierarchical rulebase shown in Figure 1.

4. Defining fuzzy operations

Although fuzzy operations definition is outside of the scope of XFL, the mechanisms for it play a fundamental role in the use of the language by any fuzzy system development tool. Essentially, any tool able to translate an XFL specification into an appropriate implementation format should use a configuration method to define (in terms of the
target language) the fuzzy operations that can be used. An XFL-based tool should support mechanisms for identifying this operations, so it can produce a complete implementation of the source specification. This section introduces the mechanism in use by the currently available tools, which we think is general and flexible enough to be extended to any target language.

In the current generation of XFL-based tools, fuzzy operations are selected to be used in an implementation by a series of directives included in the source file. These directives identify an operation definition in the configuration file used by the tool. These directives are:

- `#and` (T-norm),
- `#or` (T-conorm),
- `#not`,
- `#implication`,
- `#composition` (T-conorm for operator c),
- `#also` (T-conorm for operator also),
- `#defuzzification`.

Each of these directives uses an identifier, corresponding to the keyword for the operation definition in the configuration file used by the tool. This directives identify an operation definition in the configuration file used by the tool. These directives are:

- `#and` (T-norm),
- `#or` (T-conorm),
- `#not`,
- `#implication`,
- `#composition` (T-conorm for operator c),
- `#also` (T-conorm for operator also),
- `#defuzzification`.

Each of these directives uses an identifier, corresponding to the keyword for the operation definition in the configuration file used by the tool. XFL-based tools should supply a set of default values for each operations. As previously noted, the case of the `#composition` directive is a little subtle: if there is no such directive in a source file, rule preaggregation is simply not used.

The configuration file consists in a set of sections, each one corresponding to a class of the defined elements, and marked by a start and an end symbol. Start symbols are: `
%Tnorm`, `
%Tconorm`, `
%Negation`, `
%Implication` and `
%Defuzzification`. The end symbol is made of the characters `%%`.

Inside each section, the definition of operations has two parts: the operation identifier (in a separate line), and the body of the definition, included below its identifier in subsequent lines that must start with a tab character. The body of an operation definition (apart from the case of the defuzzification methods, that use an specific format) must contain the body of a native construct of the target language (function, procedure, macro, etc.) returning the result of the operation. Defuzzification method definitions are more complex: the definition body lines are preceded by symbols indicating the requirements imposed by the method on the membership functions for being applicable, and the point of the defuzzification process in which the contents of the lines are used. These symbols are:

- `#requires`, followed by a (comma separated) list of membership functions classes for which this method is applicable. If this symbol is not used, the method is applicable for any class.
- `#init`, which marks the start of the target language lines used for variable initialization.
- `#exit`, which marks the start of the target language lines used at the end of the process for assigning the final output value.
- `#numloop`, which indicates that the lines below are going to be executed in a loop through the universe of discourse for the variable. In this loop the value of the variable and the value of the global output membership function can be accessed.
- `#linloop`, which indicates that the lines below are going to be executed in a loop through the linguistic labels defined for the variable. In this loop the activation grade for each rule (or membership function) and the parameters of membership functions can be accessed.
- `#end`, which marks the end of the above constructs.

5. XFL-based tools

The use of XFL has allowed the creation of a set of tools which take advantage of its characteristics for the development and implementation of fuzzy logic based systems. The features of XFL have made possible tools oriented towards both hardware and software implementations and let us envisage (in a near future) new tools covering other aspects of fuzzy systems development.
The kernel of all these XFL-based tools is made of a set of functions (called the XFL-library) which provide parsing and semantics-checking facilities for specifications, and store them in an abstract syntax tree that can further be accessed by the different modules composing XFL-based tools.

5.1. Xfc: An XFL to C compiler

_xfc_ takes an XFL file and produces a C source file which implements the fuzzy inference engine defined in XFL as a C function. This file can be freely combined with any other modules to produce programs containing fuzzy logic capabilities. The only requirement imposed by the C output file derived from _xfc_ is that it must be compiled using the _libm.a_ standard library, since it uses specific mathematical functions.

The interaction between the fuzzy inference engine and the calling module is accomplished by the parameters corresponding to the input/output variables defined for the fuzzy system. These parameters belong to a specific C type (FUZZY), defined by means of a typedef construct in the output file. This type is user-configurable and any C type able to hold a numeric value can be used. Parameters are passed to the fuzzy inference engine function by reference (type FUZZY *).

_xfc_ works with a set of predefined fuzzy operations, which it applies when the source file does not include any of the directives described in section 4. These operations (which can also be explicitly referenced) are: The T-norm _sd_min_ (minimum), the T-conorm _sd_max_ (maximum), the negation _sd_not_ (complement to 1), the implication function _sd_min_ (minimum, again), and the defuzzification method _sd_CoA_ (center of area).

5.2. Xfbpa: Backpropagation learning for xfl-based systems

_xfbpa_ automatically tunes the parameters of an XFL-based fuzzy system by means of backpropagation. The error function computes the deviation of the actual behavior of the system from its ideal behavior, in terms of a set of input/output pairs. The error function used by _xfbpa_ is:

\[
E = \frac{1}{r} \sum_{i=1}^{r} \sum_{j=1}^{n} w_j \left( \frac{y_{ji} - Y_{ji}}{\text{range}_j} \right)^2
\]

where \( r \) is the number of training data, \( n \) the number of outputs from the system, \( w_j \) the weight for the output variable \( j \) in the error function (1/n by default), \( y_{ji} \) the actual output value \( j \) produced by the system for the input \( i \), \( Y_{ji} \) the ideal value for output \( j \) when input is \( i \), and \( \text{range}_j \) the range of output \( j \). This error function represents the weighted mean square deviation for the outputs with respect to their valid ranges and always takes values between 0 and 1.

_xfbpa_ allows the user to select which parameters of the system are to be learned and to tune the error function, giving more weight to those output variables considered more important in the system deviation. Furthermore, it performs a rule pruning algorithm, deleting those rules whose activation grades fall under a (user adjustable) threshold. If during the learning process rule pruning has been performed, a new learning loop is initiated upon completion of the original one, since the modification of the rulebase can interfere the learning results. The tool requires the specification of one of four possible end conditions: maximum number of iterations, minimum value for the mean deviation, minimum value for the maximum deviation or minimum value for the relative error increment. The learning process stops when one of these limits is reached.

Although one of the original objectives of XFL was the complete independence of the fuzzy operations in use, the algorithm used by _xfbpa_ does not fully allow this flexibility. Despite this, the current version of the tool support 4 T-norms, 5 T-conorms, up to 10 implication functions and 9 different defuzzification methods. The tool does not impose any limit in the complexity of the system under learning.

5.3. Xfsim: Building simulations for XFL-based Systems

_xfsim_ allows the integration of one or more XFL-based specifications in a closed-loop system simulating the global behavior of the fuzzy module(s) under evaluation and their environment. The input to this tool is a file specifying: 1) The variables that define the state of the closed-loop system; 2) The behavior of the system under simulation, by means of the interaction of different modules, which can be XFL-based fuzzy specifications, procedures with a C interface, or data directly read from a file; 3) The desired results, specified as outputs from the simulation; and 4) A set of parameters controlling the building and execution of the simulation process.

From these definitions, _xfsim_ produces a C source file which holds the main function of the simulation program and a _Makefile_ for building the simulation executable through the standard program _make_.

The simulation main function performs the following steps:

- System variables are initialized using either the values supplied by the user for the particular execution, or the default values provided by the simulation definition. The variables automatically supplied by _xfsim_ (_t_ for the simulation time and _n_ for the number of iterations in the simulation loop) are initialized as well.
• While the (user-configurable) end condition is not reached, the simulation loop is executed. At the beginning of this loop, the values of those variables selected for each output file are stored in the format specified in the definition. The rest of the loop consists in successive calls to the procedures defining the behavior of the system. System variables are passed to the procedures by reference, so any value defining the state of the system can be accessed and changed by these procedures.

• When the simulation loop has finished, output files are closed and (if the definition states so) viewer programs for the output files are started. Currently, xfsm supports two output formats: a file format (data is stored in successive lines in a text file, with some comments) that can be viewed by means of a text editor, and a gnuplot compatible format whose viewer is, obviously, the program gnuplot.

5.4. Other tools under development

Apart from the tools described in the previous section, there are some other XFL-based tools under development in the field of the XFL translators to hardware implementation languages.

xfpg produces a C source file able to generate a boolean function definition of the behavior of a XFL-based system. It integrates the fuzzy inference engine produced by xfc with other functions which provide inputs to the inference engine and give boolean format to all the (input/output) variables of the system. Once compiled, the C source file generated by xfpg produces the boolean function definition in an espresso [10] PLA compatible format. This allows the use of widely available tools for further minimization and implementation of the results.

xfvh, which produces a VHDL definition from a XFL specification. Currently an specific VHDL output generator is available [11]. Basically, it produces a VHDL entity able to be integrated with other similar elements in the definition of a more complex system or to constitute a system by itself.

All these tools are being integrated in a common graphical interface for the development of fuzzy systems, called Xfuzzy 2.0. Plus integrating the functionality of all the tools mentioned in this paper, Xfuzzy offers graphical facilities for working with XFL.

6. Conclusions

This paper presents the main aspects of XFL, a flexible and powerful language for the definition of fuzzy systems. Its more relevant features are the ability to define potentially complex systems and the independence from concrete technological aspects. We think that these features allow the use of XFL in a wide range of applications of fuzzy logic and in the study of the methods of fuzzy logic itself (connectives, defuzzification methods, etc.). The applicability of XFL in different domains has been demonstrated by the tools based on it and we believe that further tools developments and the use of the language in more applications will enrich its capabilities and expressiveness.

References