Fuzzy Systems and Neural Networks XML Schemas for Soft Computing*

A.R. de Soto, C.A. Capdevila and E.C. Fernández
Escuela de Ingenierías Industrial e Informática
Universidad de León. 24071 León, España
ddears@unileon.es, *conrado.andreu@wanadoo.es*, *ddeecf@unileon.es*

Abstract

This article presents an XML[2] based language for the specification of objects in the Soft Computing area. The design promotes reuse and takes a compositional approach in which more complex constructs are built from simpler ones; it is also independent of implementation details as the definition of the language only states the expected behaviour of every possible implementation. Here the basic structures for the specification of concepts in the Fuzzy Logic area are described and a simple construct for a generic neural network model is introduced.

Keywords: Soft Computing, Fuzzy Systems, XML, Software Components, Neural Networks.

1 Introduction

iXSCL, which stands for Extensible Soft Computing Language, is an XML vocabulary for the specification of common objects in the Soft Computing area. The first version of the language[5] only considered Fuzzy Systems[13][12] and described concepts like linguistic variables, fuzzy rule bases or fuzzy rule systems. One of the objectives of the design is to facilitate the addition of constructs from other areas; in this article a generic neural network model is defined and added to the language.

Possible applications include the integration of soft computing techniques into web resources and the exchange of specifications in a common XML based format; the EDAB project, for example, produces iXSCL specifications of fuzzy rule systems using extraction algorithms that can be later translated into software components for the Enterprise Java Beans Platform.

XML techniques used in this project include XML Schemas[3] and XSL Transformations[1]; the former define a class of XML documents, much like old DTDs

* This work has been partially supported by project TIC2000-1420 of the Spanish National R+D+I Plan and project 2002/29 of the Regional Spanish Government, Junta de Castilla y León

did, but introduce interesting Object Orientation and Relational Data Bases concepts. XSLT are transformations between documents. Their most common use is the visualization of XML data, but they are here applied to the validation of specifications. Although XML Schemas include some constructs to express conditions like uniqueness or existence, XSLT has more expressive power.

The Fuzzy Systems related part of this project owns a lot to the XFL3 language[9]. A related work on the application of XML to Fuzzy Systems can be found in[10] and other specification languages in fuzzy logic include FTL[11] and Fril[4].

The next section introduces the iXSCL language and comments on its main constructs. The discussion on Fuzzy Systems is supported by some UML diagrams that model the concepts and small fragments of sample documents; it starts with the definition of *linguistic variable types*, *rule bases* and *bindings*, which are then used to construct a *fuzzy block*. Then *map* and *layout* are introduced as key concepts for composition and for the extension of the language; the *defuzzification block* is shown as an example of a new building block. Finally, a generic model for a neural network is described and encoded to provide another type of block.

The last sections contain some comments on the extension of the language and implementation independence, which are both objetives of the design.

2 The iXSCL Language

Our main objective is to provide an implementation independent vocabulary for the specification of the common objects found in the Soft Computing area. Syntax and semantics are defined using XML techniques such as XML Schemas and XSL Transformations, and formal documentation provides the missing information. This completes the definition of the language.

Every processing of iXSCL documents must follow that definition, which states nothing about implementation details such as programming languages or software architecture. We could have an Enterprise Java Beans component which, given some data, produces a definition of a fuzzy rule system; another application, running in another platform and developed in C++, could use that definition to provide execution of the fuzzy rule system.

The whole design adopts a compositional approach; 'fuzzy rule bases', 'bindings' and 'linguistic contexts' are combined to define a 'fuzzy block', which in turn can be considered a primitive building block mapping a set of inputs to outputs. These blocks are then used again to compose other components, like a 'fuzzy rule system'.

2.1 Fuzzy Systems

2.1.1 Linguistic Contexts

The language includes the *linguistic variable* and *crisp variable* types. These are defined into *linguistic contexts* which act as name spaces; contexts are themselves organized into a tree–like structure. A simple addressing scheme is used in iXSCL documents to locate variable types, following a file system metaphor in which contexts act as directories and types as files; for instance, an attribute

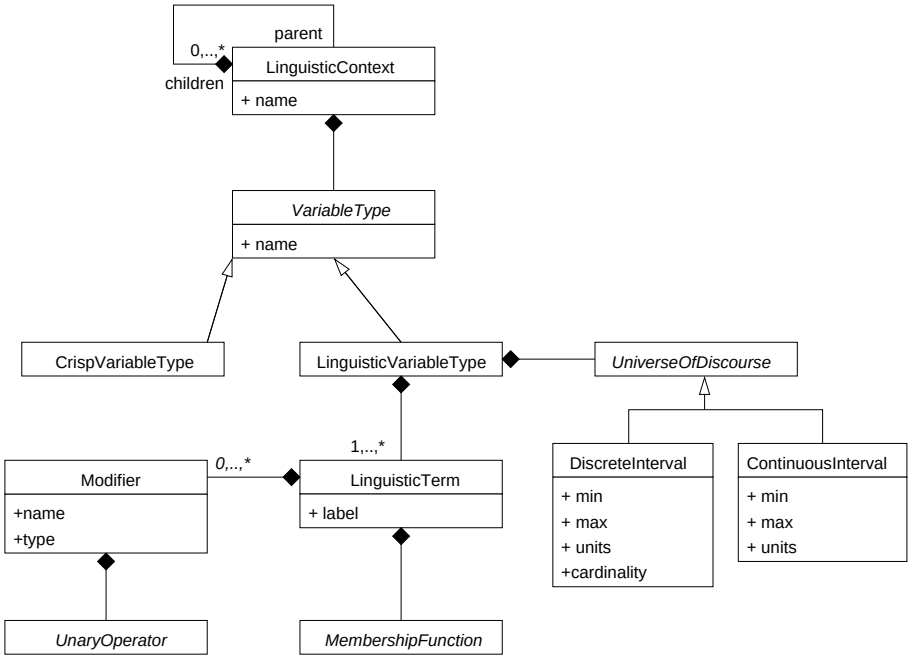

Figure 1: Linguistic Contexts

"/box/corner/Pressure" could represent the "Pressure" variable type in context "corner", inside context "box".

Crisp variable types represent values of common types found in programming languages, like floats, integers and enumerations.

Linguistic variable types carry an associated universe of discourse and one or more linguistic terms. The first version of the language considered two types of universes of discourse, continuous intervals on the set of real numbers and their discrete counterparts for a given cardinality.

Each linguistic term has a label, a membership function and optional linguistic modifiers. Modifiers are classified into *internal* and *external*[6] and defined using unary operators. For example, the operator *power* ($f(x) \triangleq x^w$) is used here to construct an external modifier *very* for label *high*:

```
<term label="high">
    <modifier type="external" name="very">
        <operator xsi:type="power">
            <w>2.018287</w>
        </operator>
    </modifier>
</term>
```

Linguistic Modifier

The XML Schema states that *power* is an *unaryOperator* which has a real

parameter, represented in XML documents as a child element *w*. Formal documentation completes the definition with the expression $f(x) \triangleq x^w$.

Using abstract types like *unaryOperator* allows for type substitution in the XML document and addition of new features[8]. A core set of membership functions and operators is defined in the XML name space of the language using abstract base types *membershipFunction*, *unaryOperator* and *binaryOperator*; XML Schema definitions include those abstract types so that instance documents can carry any derived type, like the *power* operator.

2.1.2 Fuzzy Rule Bases

A fuzzy rule base represents part of the knowledge associated with a set of fuzzy rules; it is an enumeration of rules which contain antecedents, consequents, connectives, etc. Rule bases are used as templates that will be later resolved into real objects to reuse the rule base for the definition of different fuzzy blocks. Parameters include linguistic variable names, terms, modifiers, connective operators and hedges.

There is one type of fuzzy rule, *absoluteRule* which carries a real valued weight, antecedent and consequent. Antecedents are made up of *compositionalUnits*: atomic fuzzy propositions, hedges and connectives. Atomic fuzzy propositions associate a linguistic term, with one or more optional linguistic modifiers applied to it, and a linguistic variable name; they represent expressions like "*X1 is very high*" or "*X2 is not dangerous*". Compositional units can themselves be modified by linguistic hedges and composed using connectives, like in "*approximately, X1 is high and X2 is near*" which uses hedge *approximately* and connective *and*. Consequents are limited in this version of the language to single fuzzy propositions and real valued expressions like "*X1 is 4.23*".

```
<rule xsi:type="absoluteRule" weight="+0.872">
    <antecedent>
        <hedge name="approximately">
            <composition connective="and">
                <proposition variable="X1">
                    <modifier>very</modifier>
                    <label>high</label>
                </proposition>
                <proposition variable="X2>
                    <label>near</label>
                </proposition>
            </composition>
        </hedge>
    </antecedent>
    <consequent>
        <expression variable="Z1">
            <value>4.23</value>
        </expression>
```

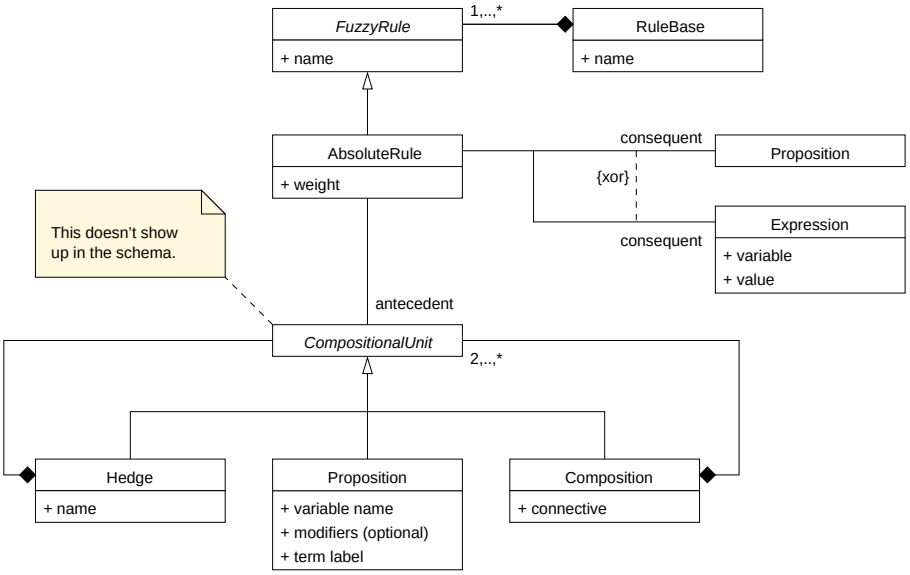

Figure 2: Fuzzy Rule Bases

```
    </consequent>
</rule>
```

Absolute Fuzzy Rule

The rule base plays the role of a template with parameters that must be associated with concrete objects; for example, suppose "*X1*" appears in the rule base associated with the term "*high*" and its modified version "*very high*". All these "*X1*", "*high*" and "*very high*" are considered just parameters. In order to obtain a working fuzzy block "*X1*" must be associated with a real linguistic variable, "*high*" with one of its terms and "*very*" with a modifier. See 2.1.5.

2.1.3 Bindings

In order to complete the specification of a fuzzy block from a rule base all the parameters must be associated with real objects; bindings provide operators for every connective named in the rule base, as well as implication and aggregation operators.

```
<bindings name="BX1">
    <connective name="and">
        <operator xsi:type="bounded-prod"/>
    </connective>
    <hedge name="approximately">
        <operator xsi:type="power">
            <w>2.442231</w>
        </operator>
```

```
    </hedge>
    <implicationOperator>
        <operator xsi:type="max"/>
    </implicationOperator>
    <aggregationOperator>
        <operator xsi:type="sum"/>
    </aggregationOperator>
</bindings>
```

Bindings

Bindings carry a different type of knowledge than fuzzy rule bases, and can also be reused to define fuzzy blocks.

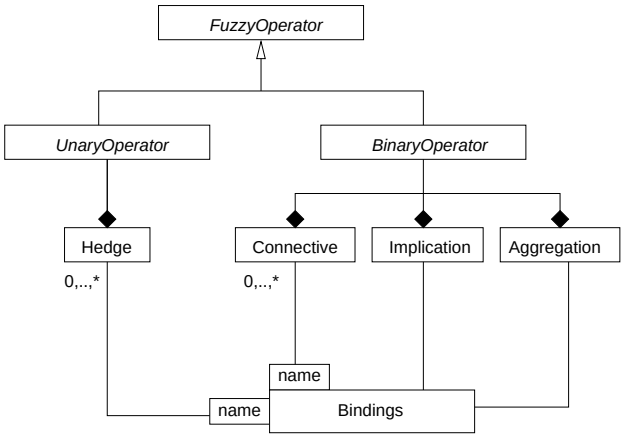

Figure 3: *Bindings*

2.1.4 Layouts

Every block that extends the abstract type *map* can be used in a *layout* to construct another block; the actual definition of a layout uses an *operation*, like "*serial*" or "*parallel*" composition to perform the mapping, but other operations representing iteration or conditional processing will be considered in next version of the language.

A layout using the "*serial*" operation takes and ordered list of maps and accepts bindings between the input of a block and the output of any of the preceding blocks. The "*parallel*" operation states that there are no bindings between its blocks and therefore they could be working in parallel. When defining new operation types, their behavior must be defined in the documentation.

The obtained layout is itself a map and can be used as a building block in another layout.

Both *fuzzy block* and *defuzzification block* are examples of maps, but other blocks can be added to the language extending the abstract base type *map*. For example, new blocks for neural networks could be combined with existing fuzzy

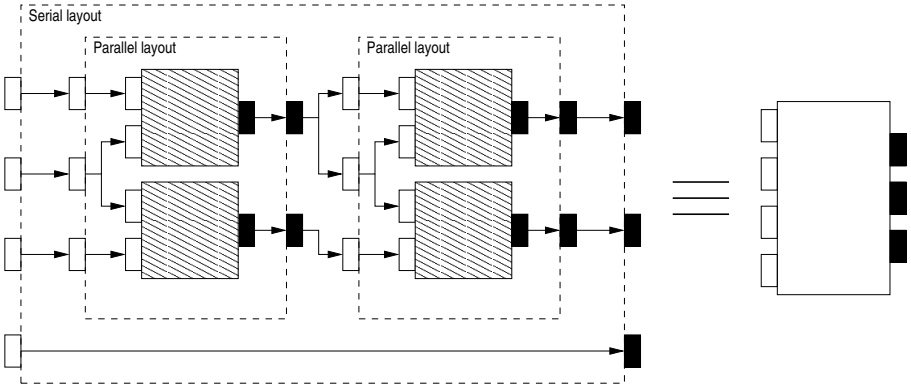

Figure 4: Internal Structure of a Simple *Layout*

logic–based ones to construct hybrid layouts; each map exposes the type of input it needs and the output it provides, thus allowing for type checking when it is bound to another component. When needed, new blocks could also be defined to adapt different variable types.

Figure 5 shows *Map* as a black box with a set of typed input and output variables. The *Layout* is a kind of map built from other maps through the application of an operation, like parallel or serial composition, that determines which bindings between outputs and inputs are legal.

```
1  <layout name="Serial Composition">
2      <parameters>
3          <input name="level-A1" type="/sensors/Level"/>
4          <input name="level-A2" type="/sensors/Level"/>
5          <input name="warning" type="/sensors/Warning-level"/>
6      </parameters>
7      <operation xsi:type="serial"/>
8      <block name="FB1" ref="FuzzyBlock-1">
9          <link from="Input/level-A1" to="X1"/>
10         <link from="Input/level-A2" to="X2"/>
11     </block>
12     <block name="FB2" ref="FuzzyBlock-2">
13         <link from="FB1/Z1" to="X1"/>
14         <link from="FB1/Z2" to="X2"/>
15     </block>
16     <block name="Output">
17         <link from="FB2/Z1" to="warning"/>
18     </block>
19 </layout>
```

Layout

This fragment defines a *layout* with two inputs, *level-A1* and *level-A2*, and one output *warning*, using a serial composition of two maps. Each *layout* has always

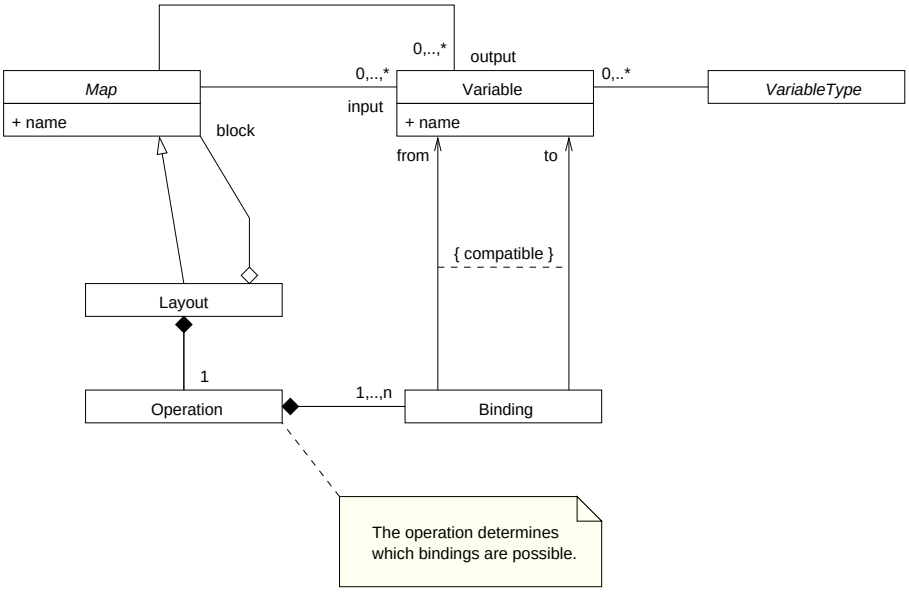

Figure 5: *Maps* and *Layouts*

two dummy maps, named *Input* and *Output*, which are used to refer the inputs and outputs of the *layout*.

2.1.5 Fuzzy Blocks

A fuzzy block results from the association of a *fuzzy rule base* template with *bindings* and *linguistic variables*; it is a complete specification of a functional block that uses linguistic variables as inputs and outputs. The XML Schema *fuzzyBlock* type extends the abstract base type *map* so that it can be used in layouts.

The fuzzy block element contains the information to substitute the parameters in its rule base with real objects: variables, terms and modifiers are explicitly associated with a type in a linguistic context; operators are associated by means of a previously defined *bindings* element.

For example, "*X1*" could be used to define input variable "*left-sensor-temperature*" of type "*/sensors/Temperature*" (context *sensors*, variable type *Temperature*) with term "*hot*" acting as "*high*" and modifier "*very*" bound to "*extremely*". This would give rules like "*if left-sensor-temperature is high then...*".

Another fuzzy block with the same rule base could be obtained by using "*/tank/WaterLevel*" and "*dangerous*" instead, to define input variable "*water-level*"; rules would then be "*if water-level is dangerous then...*".

The same procedure must be applied to hedges and connectives, to resolve "and", for instance, into a fuzzy operator and the hedge "approximately" into another, using *bindings*.

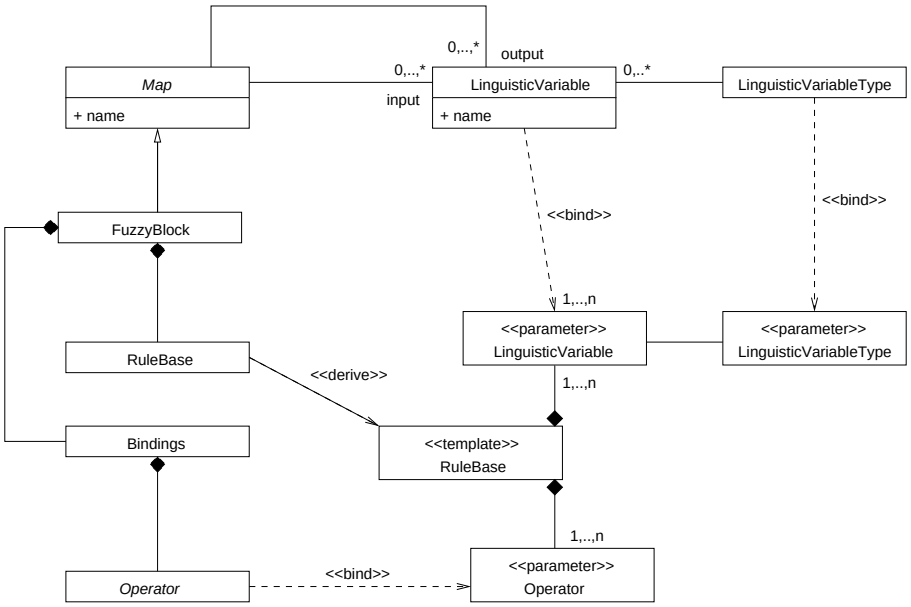

Figure 6: Fuzzy Blocks

```
<fuzzyBlock name="FB1"
      bindings="BX1"
      ruleBase="RB1">
      <parameters>
            <input name="left-sensor-temperature"
                  type="/sensors/Temperature"/>
            <input name="right-sensor-temperature"
                  type="/sensors/Temperature"/>
            <output name="blockage"
                  type="/car/wheels/Blockage"/>
      </parameters>
      <bindings>
            <variableParam parameter="X1"
                  value="left-sensor-temperature">
                  <termParam parameter="high"
                        value="hot">
                        <modifierParam parameter="very"
                              value="extremely"/>
                  </termParam>
            </variableParam>
            ... other variables ...
      </bindings>
</fuzzyBlock>
```

Fuzzy Block

2.1.6 Defuzzification Blocks

The composition of blocks associates inputs and outputs; for example, a *layout* with a *serial* operation on two fuzzy blocks ties the inputs of the second block with outputs from the first one, but this should only be allowed when the types involved are compatible. The language can be extended to include other variable types and blocks, and many of these blocks will work on crisp inputs. The *defuzzification block* applies defuzzification operators to a set of linguistic variables to obtain crisp values; it is used as an adapter between fuzzy outputs and crisp inputs.

The *defuzzification block* extends the abstract type *map* and is defined by a list of bindings: input, defuzzification operator and output. A default operator can be included.

2.1.7 Fuzzy Rule Systems

The fuzzy rule system is finally defined by either a layout combining one or more fuzzy blocks or a single fuzzy block. Once again the result is a mapping from inputs to outputs, but it comes closer to the idea of a working, self–contained unit that could be translated into a software component.

2.2 Neural Networks

The abstract base type *map* provides the common view of a processing block with typed inputs and outputs; some examples of such blocks are the *fuzzy rule system* and *defuzzification block*. Here a generic neural network is defined as a new type of map, thus adding a new building block to the language. Both inputs and outputs of the network are considered vectors of real numbers, of *vectorOfReals* variable type.

The model describes the architecture of a neural network as a set of interconnected layers, and each layer as a set of interconnected processing nodes. The whole network computes until a given stop criterion is met, like reaching a maximum number of iterations or stability; each layer works either in *synchronous* or *asynchronous* mode. All the nodes in a *synchronous* layer are computed at the same time, and therefore their outputs are not propagated to other nodes inside the layer untill the whole computation is done. In an *asynchronous* layer all the nodes are fired in random order, and their outputs taken into account for the next node inside the layer. For example, layers in an MLP (multilayer perceptron) can be either synchronous or asynchronous, as there are no connections between nodes in each layer, but a layer in a Hopfield network would yield different results for each mode. Processing nodes defined so far include artificial neurons and bias nodes.

This model does not address operations like training or testing the network, it only includes the necessary information to compute its outputs given a set of inputs.

2.2.1 Layers and Processing Nodes

The iXSCL schema defines two processing node types: the classic artificial neuron[7] and bias nodes. The *nonLinearNeuronNode* carries an activation function, weights and optional named tags. Tags can be used to associate attributes to the neuron, like a label in a self–organizing map. Bias nodes provide a constant output of a given real value.

Nodes are defined inside a layer, which also describes the set of connections between them. A *signalFlow* element declares where are the inputs of every node taken from and which nodes provide the outputs of the layer; a simple syntax is used in which *iN* represents the n–th input of the layer, *oN* its n–th output, and the *id* of a node its output. The whole layer is given a *synchronous* or *asynchronous* model with a *model* attribute.

```
<layer id="lattice" nInputs="3" nOutputs="3" model="asynchronous">
    <processingNodes>
        <node id="n1" xsi:type="nonLinearNeuronNode" nInputs="5">
                <activationFunction xsi:type="thresholdActivationFunction">
                    <min>0.0</min>
                    <max>1.0</max>
                </activationFunction>
                <weights>
                    0.123124124 0.259528285 0.87721231
                    0.1928123 0.12938241
                </weights>
                <tags>
                    <tag name="label">Class A1</tag>
                </tags>
        </node>
        <!-- other nodes: n2, n3 -->
    </processingNodes>
    <signalFlow>
        <inputs node="n1">i1 i2 i3 n2 n3</inputs>
        <inputs node="n2">i1 i2 i3 n1 n3</inputs>
        <inputs node="n3">i1 i2 i3 n1 n2</inputs>
        <outputs>n1 n2 n3</outputs>
    </signalFlow>
</layer>
```

Layer

2.2.2 Networks

In order to describe a neural network a set of layers is defined, and connections between them declared using a *signalFlow* element. This element is similar to the one inside each layer, but here *name:oN* represents the n–th output of the *name* layer. The description is completed with a stop criterion which can be either a fixed number of iterations or stability.

```
<neuralNetwork name="Hopfield">
    <parameters>
        <input name="input-vector" type="/sensors/NoiseSample"/>
        <output name="output-vector" type="/levels/NoiseLevel"/>
    </parameters>
    <networkLayers>
        <layer id="lattice" nInputs="3" nOutputs="3">
            <!-- definition of this layer -->
        </layer>
    </networkLayers>
    <signalFlow>
        <inputs node="lattice">i1 i2 i3</inputs>
        <outputs>lattice:o1 lattice:o2 lattice:o3</outputs>
    </signalFlow>
    <stopCriterion xsi:type="stability"/>
</neuralNetwork>
```

Network

3 Extensions

As the language targets an active area of research some techniques from XML Schemas, like name spaces and abstract types, are used to facilitate extensions[8].

Consider the addition of a binary operator; the new XML Schema type must extend the abstract base type *binaryOperator* in the iXSCL name space and reside in its own name space; the content model will include the XML needed to define an instance of this operator, and the associated documentation will state the formal definition of the operator and its parameters.

4 Implementation Independence

By choosing XML we get a platform neutral language, based on accepted open standards. XML Schemas and XSL Transformations are both core technologies of XML that are used here to define and validate specifications, and we already have the tools to access its contents. We need a way to complete the definition of the language in order to avoid misinterpretations.

We are using formal documentation that completes the model contained into the schemas and transformations. Consider, for example, a triangular membership function associated with a linguistic term; both schemas and transformations will be used to assert that every instance of this construct has three elements, named 'a', 'max' and 'b', containing ordered real numbers. Any implementation of the language must be able to verify this and extract the associated values, and it will probably do so using some standard API like DOM or XPath; what a 'triangular membership function' is and the correct interpretation of those values to define it are given in the formal documentation.

5 Conclusions

The iXSCL language definition is not closed; future versions will improve it and include objects from other areas. The management of versions is a problem that must still be approached; even though extensions are possible the language should keep a meaningful and stable core.

XML is gaining ground and there is a lot of ongoing work to produce software tools that take advantage of it, many using Java and thus being more platform independent.

As we have mentioned before, more than one implementation of the iXSCL language is possible, and XML documents are platform independent. This facilitates the integration of heterogeneous distributed systems, from lightweight mobile clients to an IDE running on a workstation.

The software of the EDAB project is an example of application for the language. EDAB seeks to provide a solution to the management of fuzzy rule systems in a distributed and heterogeneous environment, and to do so through software components. It contains, for example, elements for the automatic extraction of fuzzy rules from data, the analysis and compilation of fuzzy rule systems specifications into a working component or the execution of queries against those components using data bases.

The project also includes the development of an IDE, where a set of JavaBeans are being developed to define specifications through a graphical interface.

iXSCL is the common language in which those software components communicate: an extraction component, for example, applies an algorithm to a set of data and produces an specification of the fuzzy rule system; this is an XML document which can be later analyzed and compiled into another software component which implements the fuzzy rule system.

References

[1] XSL transformations (XSLT) version 1.0. Technical Report REC–xslt–19991116, W3C (MIT, INRIA, Keio), November 1999.

[2] Extensible markup language (XML) 1.0 (second edition). Technical Report REC–xml–20001006, W3C (MIT, INRIA, Keio), October 2000.

[3] XML schema part 0: Primer. Technical Report REC–xmlschema–0–20010502, W3C (MIT, INRIA, Keio), May 2001.

[4] J.F. Baldwin, T.P. Martin, and B.W. Pilsworth. *Fril – Fuzzy and Evidential Reasoning in Artificial Intelligence*. Research Studies Press Ltd., 1995.

[5] A.R. de Soto, Conrado A. Capdevila, and Eva Cuervo Fernández. An xml vocabulary for soft computing. In *Third EUSFLAT Conference*, pages 645–650, Sep 2003.

[6] A.R. de Soto, E. Trillas, and M.J. Herrero. Antónimos y modificadores lingüísticos. In *VII Congreso Español sobre Tecnologías y Lógica Fuzzy*, pages 7–14, Sep 1997.

[7] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall International, Inc., 1998.

[8] The MITRE Corporation and the xml-dev list group, http://www.xfront.com-/BestPracticesHomepage.html. *XML Schemas: Best Practices*, 2002.

[9] F.J. Moreno-Velo, S. Sánchez-Solano, A. Barriga, I. Baturone, and D.R. López. A specification language for fuzzy systems. *Mathware and Soft Computing*, VIII(3):239–253, 2001.

[10] Klaus Turowski and Uwe Weng. Representing and processing fuzzy information — an XML–based approach. *Knowledge–Based Systems*, 15(1–2):67–75, 2002.

[11] Constantin von Altrock. *Fuzzy Logic and NeuroFuzzy in Business and Finance*. Prentice Hall, 1997.

[12] Lotfi A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC–3(1):28–44, January 1973.

[13] Lotfi A. Zadeh. Soft computing and fuzzy logic. *IEEE Software*, 11(6):48–56, November/December 1994.