J. of Mult.-Valued Logic & Soft Computing, Vol. 15, pp. 209–228
Reprints available directly from the publisher
Photocopying permitted by license only

Formal Specification under Fuzziness

V. LÓPEZ[1] AND J. MONTERO[2]

[1]*Department of Computers Architecture, Complutense University, Madrid, Spain*
E-mail: vlopez@fdi.ucm.es
[2]*Faculty of Matmematics, Complutense University, Madrid, Spain*
E-mail: javier_montero@mat.ucm.es

Received: April 27, 2007. Accepted: March 23, 2008.

Judging the quality of any decision making procedure is a key problem whenever there is no possibility of developing a sequence of experiments allowing some kind of ratio relative to good results. It may be the case that we have only chances for a unique experiment or no similar experiences are available, but it may be also the case that no standard experiment allows the observation of such a good behavior, simply because such good behavior can not be properly defined, in terms of standard *crisp* experiments. This situation is quite often associated to complex decision making problems. Then the only support we can find for our decision is the decision process itself, i.e., the consistency of the arguments leading to such a decision. Checking the quality of such a procedure becomes in this framework a key issue. Since specification is being quite often poorly defined, we postulate that the design and formal specification of algorithms and processes require a fuzzy approach.

Keywords: Fuzzy logic, formal specification, decision making.

1 INTRODUCTION

According to the *1995 Standish Chaos* report [26], 50% of all project failures are caused by requirements problems (requirement analysis, specification and high-level design). But addressing these requirement problems imply to combine formal methods with other approaches, if we really want to build a real system (see Hall [9]. In this paper we stress how important is to approach natural language to formal specifications, and in this context we show a particular way in which fuzzy logic can help certain areas within software engineering [27]. Of course, fuzzy logic has already been applied to several areas of software engineering (see, e.g., [7, 12, 21], but mainly from a database viewpoint. Our approach focusses on the key role fuzzy approaches may play on formal specification, as already pointed by the authors in the 10th Joint Conference on Information Sciences [16].

As soon as description of objectives and requirements of a software process are poorly defined in nature, even in the case that a procedure apparently fits them, formal specification can not be properly developed with standard techniques. If a set of previous experiences are available or can be developed, additional studies can perhaps allow some statistical procedure in order to evaluate the quality of the procedure. Still, we may have serious problems in case the output needs a linguistic translation where a probabilistic model may not properly fit, see [20]. In practice, most procedures related to human decision making require ill defined information both in the input and the output. So, we need alternative approaches in order to improve software specification, and more specifically, formal specification of critical algorithms and systems in the software process. As pointed out by Sommerville [24], natural language is often used to write system requirements specifications as well as user requirements. However, system requirements are more detailed than user requirements, and then natural language specifications can be confusing and hard to identify. We can find in the literature formal methods for assuring the right behavior of software. Due to their cost, formal methods are more popular in critical processes. For example, security systems will be an important area for formal methods, see [8]. In addition, decision makers use to support their decisions on certain information to be computed by *ad hoc* algorithms, see [19]. Naturally, part of the input data comes from a linguistic framework, perhaps fuzzy relations or fuzzy properties, which should be de-fuzzyfied before execution. Hence, from the above arguments we can conclude the interest of introducing fuzzy logic methodologies for an accurate formal specification of quite a number of procedures as far as natural language of human beings is being involved. In particular, pre and postcondition sentences can be written by means of a propositional logic in which fuzzy sets an fuzzy relations are added.

Moreover, software engineering is quite often based upon documentation methods where information is given in natural language, which implies ambiguity and imprecision. Software engineering should in this case try to improve the quality of software by means of diagrams, pictures and a large amount of text.

In this paper we continue the approach initiated by the authors in [16], taking *Z notation* [29] (a formal method that solves ambiguity by using crisp set theory and mathematical logic) as the initial model and then bringing fuzzy logic into specifications in order to develop formalizations for algorithms and software specifications, increasing its ability to deal with ambiguities produced by natural language (those specifications involving fuzzy constrains and properties should be approached within a fuzzy framework). This paper points out the future relevance of this kind of specifications in software specification of requirements, which should become a key element in any software project involving non-technical users.

2 FORMAL SPECIFICATION OF ALGORITHMS

Since an algorithm **A** can be formally described by its formal specification, such a formal specification will describe the computational situation before and after executing an algorithm, by means of relationships between the input and the output data. In many cases, programmers use to decide the good behavior of an algorithm at a glance. After all, programmers have their own experience and they act as experts making decisions based on their common sense. But such a procedure will be acceptable in those cases in which the algorithm is very simple, or goals are very simple [9]. However, most algorithms are not in this cluster. First of all, the code is usually written after an agreement about the aim of the process that is going to be written. Analysts, designers and programmers are involved into developing and programming, and all of them will share those documents that model the business process and system requirements. It is sometimes difficult to use the language in a precise and unambiguous way without making the document wordy and difficult to read.

2.1 A first approach to fuzzy specification of algorithms

We should realize that quite often some parts of the specifications are expressed in natural language [24]. We speak then about informal specifications. Hence, any computation will require a previous formalization (formal specifications). Natural language uses a lot of fuzzy relations and perceptions that we can not avoid, either in the input data (precondition) nor the output data (postcondition).

The following definition generalizes the concept of formal specification [10] of an algorithm in a fuzzy framework, see [16].

Definition 2.1. Let **A** be an algorithm or a process. The 'fuzzy formal specification' of **A** is given by a triple $(ED, fPre, fPost)$ where

(1) *ED* is an explanatory database in the sense of [31], that is, a collection of relations (fuzzy and crisp) in terms of which the meaning of *fPre* and *fPost* are defined. It contains also the universe of discourse and whatever function that the goals need to be explained;

(2) *fPre* (fuzzy precondition) is the canonical form in the sense of [31] of propositions and fuzzy first order formulas that expresses the initial constraints and knowledge about input data; and

(3) *fPost* (fuzzy postcondition) is the canonical form after execution of the algorithm (fuzzy constraint propagation and inference is due, and the postcondition is the formalized goal of the algorithm or the process).

The following example shows the difference between evaluations in fuzzy and crisp framework, see [15].

Example 2.1. Let X be a subset of n members in a universe U of apples, and that we should decide if '*most*' apples in X are '*sweet*'. Modeling this problem in a crisp environment may not be accurate, since fuzzy interpretation of '*most*' or '*sweet*' is closer to the natural language.

Alternative crisp and fuzzy specifications for this example are showed below. The vector X contains the numeric characteristic about the taste of each member. The 'l' variable is a value assigned by an expert in order to decide if someone is '*sweet*' enough. Crisp specification develops a Boolean output b being '$True'$ if and only if there are more than $ndiv2+1$ members being '*sweet*'. Otherwise the output b will be '*False*'.

```
fun Crisp (X: array [1..n] of real;
                l: real)    out b: boolean;
{$Pre:\ n \geq 0$}
{$Post:\ b =$
$[(\sharp i \in \{1\ldots n\} : (X[i] > l)) > n\ div 2 + 1]$}
```

But a fuzzy specification defines the output as a fuzzy set, whose membership degree is a real number b between 0 and 1, representing the degree of truth of '*most apples are sweet*'. This meaning is implemented by the fuzzy set $\mu_{mostXareSweet}$. In order to do that, this fuzzy specification uses two important fuzzy sets as input: the fuzzy set '*most*' (μ_{most}) and the fuzzy set '*sweet*' (μ_{sweet}).

```
fun Fuzzy ( X: array[1..n] of real;
        $\mu_{most} : [0,1]^n \to [0,1]$,
        $\mu_{Sweet} : [0,1] \to [0,1]$ ) out b: [0,1];
{$fPre:\ n \geq 0$}
{$fPost:\ b = \mu_{mostXareSweet}(X)$
        $= \mu_{most}(\mu_{Sweet}(x_1), \ldots, \mu_{sweet}(x_n))$}
```

Now, let us assume the following explanatory database, where SWEET and MOST are fuzzy sets. Taste is a numeric variable and ArrayN is an array of size N.

```
ED = POPULATION [Name, Taste]+
SWEET[Taste; $\mu_{sweet}$]+
MOST[ArrayN; $\mu_{most}$]
```

The constrained variable in $\{fPre\}$ and $\{fPost\}$ is the array Taste, which in terms of ED may be expressed as

$$X =_{Taste}\ POPULATION[Name = Name]$$

Since the field *Name* is not instantiated, X became a list instead of a single number. We can consider, for instance, the following output

$$X = (175, 176, 180, 160, 160, 160, 174, 174, 174, 174, 173)$$

and a level $l = 175$. In this case the crisp algorithm will compute the output value ($b = \mathit{FALSE}$) as a result of the following evaluations:

i	$X[i] > l$
1	$175 > 175 \equiv \mathit{FALSE}$
2	$176 > 175 \equiv \mathit{TRUE}$
$\dots$	$\dots$
11	$173 > 175 \equiv \mathit{FALSE}$

Notice that the crisp expression

$$[\sharp i \in \{1..n\} : (X[i] > l) > n \; \mathit{div} \; 2 + 1]$$

is instantiated to

$$2 > 6 \equiv \mathit{FALSE}$$

Experts will then add definitions to the above fuzzy model, related to fuzzy properties and fuzzy relations.

For example, lets assume the following definition for property '*sweet*':

$$\mu_{sweet} : \mathbf{R} \rightarrow [0, 1]$$

where

$$\mu_{sweet}(x) = \begin{cases} 0 & x < l - 5; \\ \sqrt{\frac{x-l+5}{10}} & x \in [l-5, l+5]; \\ 1 & x > l + 5. \end{cases}$$

Applying this function to every data point in X, the following set of degrees is obtained:

$$\widetilde{X} = (0.70, 0.77, 1.0, 0, 0, 0, 0.63, 0.63, 0.63, 0.63, 0.54)$$

where

$$\widetilde{X}[i] = \mu_{sweet}(X[i]), \quad \forall i \in \{1 \dots N\}$$

Lets then assume that '*most*' can be modeled as a function assigning a degree of truth about a given property within a family of the above degrees,

$$\mu_{most} : \mathit{ARRAY}[1..N] \rightarrow [0, 1]$$

where

$$\mu_{most}(\widetilde{X}) = \frac{\sum_{i \in \{1..N\}} \widetilde{X}[i]}{N}$$

Such an interpretation will be stored in the explanatory database and will provide a truth degree of the fuzzy postcondition

$$\begin{aligned}\mu_{mostXareSweet}(X) &= \mu_{most}(\widetilde{X}) \\ &= \mu_{most}(\mu_{sweet}(X[i]) : i = 1 \dots n) \\ &= 0.5027\end{aligned}$$

Other aggregation operators (see, e.g., [4,5]) can be considered in order to define the fuzzy set '*most*', for example by means of the median instead of the average:

$$\mu_{most}(\widetilde{X}) = Median(\widetilde{X}) = 0.63 = \mu_{mostXareSweet}(X)$$

In any case, comparison between crisp and fuzzy specifications is extremely relevant: Note how in this example crisp specification is driven towards a 0 (-*FALSE*-), while fuzzy specification suggests a degree of truth around 63 percent. Both results are dramatically different.

Fuzzy specification of algorithms and processes can be therefore understood as a way to formalize constrains and requirements of a problem.

2.2 Some alternative approaches

In [1] it is defined a program as a set of rules, each one annotated by a truth degree, in such a way that queries to the system are goals that are introduced as sets of atoms, linked with aggregation operators, see [4]. In particular, [1] presents programs that are written in Prolog, so they can develop tools that translate fuzzy logic program (in the sense of logic programming) into Prolog code. Although it is a good way to approach the fuzzy logic programming by means of logic programming, this is not the only way to capture information from natural language and involve it within a program or a software project.

Still, we want to we emphasize the postcondition goal, and try to expand to all areas of computing the knowledge and perceptions that natural language provides us. In particular, functional languages are open to fuzzy logic. The high order and the lambda calculus [3] are tools that allow us to set programs (or functions) with fuzzy specifications in the sense given above. High order programming use functions as values, it can pass functions as arguments to other functions, and functions can be the return value of other functions. In addition, functional languages, as Haskell succeed in developing tools for fuzzy logic evaluations, see [14]. As one may check in the code, function `evalR` has some other functions as arguments and is able to compute efficiently the truth value of a first order formula in a fuzzy semantics given as a set of functions: T-norm, T-conorm and consistent negation, see [4].

Although high order style of programming is mostly used in functional languages, it can be very useful in object-oriented languages as C++ or Java. For example, Xfuzzy [30] uses fuzzy technologies in modelling specification (inference rules and properties), and produces Java code that can be perfectly integrated into our project. Moreover, languages as C++ and Java include '*assertions*' as a methodology that is provided within the programming language to test the correctness or assumptions made by the program. Both languages define an assertion as a statement containing a boolean expression that the programmer believes to be true at the time the statement is executed, see [2]. The use of assertions was introduced in specifications from the ages of the Eiffel programming language [18]. Assertions and exceptions derive themselves from de concept of '*Design by Contract*', which is followed in most of the object oriented languages.

The design by contract theory states that an application is implemented or designed according to a contract or specification agreed upon by the developing team and the client about goals and behavior of the application. Assertions can be efficient tools to ensure correct execution of a program. They improve the confidence about the program assuming a correct way of use [2].

An assertion statement can be written in two forms:

```
assert Expression;
```

and

```
assert Expression1; Expression2;
```

The first one evaluates the boolean form '`Expression`' and the assertion fails if it returns false. The expression here contains information about the system that must be true in a correct execution. The second form evaluates the boolean expression '`Expression1`' and executes the action in '`Expression2`' if it fails. Usually, 'Expression2' returns a value that informs about the error.

All assertions must be true while the program is running, otherwise the program is failing and the failed assertion acts as token of the error. Assertions can be used within programs in the code as preconditions, postconditions, internal invariants in the classical way, and also class invariants in object-oriented programming (a class invariant specifies the values of attributes across multiple instances of a class).

2.3 A new approach to fuzzy specification

Despite of all advantage of assertions, a fuzzy context works with fuzzy relations that became fuzzy assertions. In this sense, expressions as arguments of an assertion statement must be evaluated into the whole range [0, 1]. The value returned then will be a truth value according to the goodness degree of the expression.

On the other hand, correctness of algorithms process is developed by means of inference rules that allow reasoning about computing states as assertions. The following definitions were given in [17] in order to establish the syntax of this kind of rules. They describe the fuzzy inference rules associated to each instruction of code. We shall be then able to prove the correctness of single parts of code, named triple's Hoare, see [6]:

$$\{Q_1\}\mathit{Inst}\{Q_2\}$$

Definition 1. A '*correctness condition*' is a first order formula or a Hoare's triple.

Example 2.2. Following expressions are correctness conditions:

- $\{x > 0\}x := x + 1\{x - 1 > 0\}$-Hoare' triple-.
- $\{x + y \leq\}$-first order logic formula, boolean condition-.

Definition 2.2. An '*inference rule*' in computer programming is a couple (*premises, conclusion*) where *premises* is a list of correctness conditions or requirements (assertions) and *conclusion* is another correctness condition (the result that the requirements ensure). Inference rules are represented as follows:

$$\frac{\{\mathit{premises}\}}{\{\mathit{conclusion}\}}$$

A situation expressed by '*conclusion*' is got if all premises are got. The meaning of 'got' here depends on the framework (crisp or fuzzy) in which we were working. In order to generalize both cases, we express the inference rules in terms of truth values of their premisses and their conclusions. This truth value will be 0 or 1 in the crisp case and whatever value within the interval [0, 1] in the fuzzy case.

The set of Hoare rules [10] of inference are used for correctness in the crisp framework. Fuzzy context requires a generalization of all Hoare rules by means of evaluation of correctness conditions, see again [17].

Let α be a goodness degree, then **FR** is defined as the set of acceptable (a truth value greater or equal to a goodness degree α) triples

$$(\{Q\}\mathit{Inst}\{R\}) \in \mathbf{FR} \Leftrightarrow \mathit{eval}(\{Q\}\mathit{Inst}\{R\}) \geq \alpha$$

Definition 2.3. Based on the Hoare inference rules, we consider the following fuzzy rules:

- Assignment ($AsfR$):

$$\frac{}{\mathit{eval}\{Q_x^{\mathit{Exp}}\} \leq \mathit{eval}\{Q\}}$$

There is no premise in this rule, so the conclusion is always true (the only constrain will always hold). $eval\{Q\}$ is the truth value of the assertion $\{Q\}$, and $eval\{Q_x^{Exp}\}$ is the truth value of the assertion $\{Q\}$ after the substitution $[x/Exp]$. This rule holds that the truth value of assertion $\{Q\}$ greater or equal to the value $eval\{Q_x^{Exp}\}$ (fuzzy framework assumes some information lose, crisp framework is static).

- Conditional (*IffR*):

$$\frac{(\{Q \wedge f\,B\}Inst1\{R\}) \in \mathbf{FR},\ (\{Q \wedge \neg_f B\}Inst2\{R\}) \in \mathbf{FR}}{(\{Q\}\text{if } f\,B \text{ then } Inst1 \text{ else } Inst2 \text{ endif } \{R\}) \in \mathbf{FR}}$$

  Similarly with the simplified rule:

$$\frac{\{Q \wedge f\,B\}Inst\{R\}) \in \mathbf{FR},\ (\{Q \wedge \neg B\} \Rightarrow \{R\}) \in \mathbf{FR}}{(\{Q\} \text{ if } B \text{ then } Inst \text{ endif } \{R\}) \in \mathbf{FR}}$$

- Sequence of instructions (*SeqInstfR*):

$$\frac{\left[\prod_{i=1}^{k}\{Q_{i-1}\}Inst_i\{Q_i\}\right] \geq \alpha}{(\{Q_0\}SecInst\{Q_k\}) \in \mathbf{FR}}$$

  where *SecInst* is the sequence of instructions $Inst_1; \ldots; Inst_k$;

- Iteration (*WhfR*): based on the classical iteration rule of Hoare [10],

$$\frac{\{Q\}Inic\{I\}, \{I \wedge \neg B\} \Rightarrow \{R\}, \{I \wedge B\}SInst\{I\}}{\{Q\}Inic;\ \text{While } B \text{ do } SInst \text{ enddo } \{R\}}$$

  lets consider

$$\alpha_1 = eval(\{Q\}\ Inic\ \{I\})$$
$$\alpha_2 = eval(\{I \wedge \neg B\} \Rightarrow \{R\})$$
$$\alpha_3 = eval(\{I \wedge B\}\ SecInst\ \{I\})$$
$$\alpha_4 = eval(\{Q\}\ Inic;\ \text{While } B \text{ do } SecInst \text{ enddo}\{R\})$$

  The inference fuzzy rule then is

$$\frac{(\bigwedge_{1 \leq i \leq 3} \alpha_i) \geq \alpha}{\alpha_4 \geq \alpha}$$

where $\alpha_1, \alpha_2, \alpha_3$ play the role of premisses. If its evaluation exceeds the goodness degree α, the rule infers that consequence evaluation (α_4) also exceeds the same degree.

These fuzzy inference rules will allow reasoning about fuzzy specifications and computing states during the execution of the algorithm. To put these rules into practice, a new assert statement is proposed, based on assertions in C++ or Java:

Definition 2.4. Let `Expression` be an assertion into the fuzzy logic, and `lab` a linguistic label

`Tlabel=[true,almost true,not false-not true,almost false,false]`

and `Expressioni`, $\forall i \in \{1 \ldots 5\}$ a value or an action. The `fassert` statement syntax is defined in the following way:

```
fassert Expression; deg; case lab:Tlabel of
                   true: Expression1;
                   almost true: Expression2;
                   not false-not true: Expression3
                   almost false: Expression4;
                   false: Expression5;
                   endcase;
```

It is therefore evaluated `Expression`, and the returned value is stored into `deg`. Then a linguistic label is assigned to variable `lab` and, finally, the corresponding `Expressioni` is evaluated or executed. The third argument (`case`) is optional and admits other ranges and default clause with `default`.

Combining evaluations with the inference rules and the assert statement is a good option to allow reasoning about computing states into a program. The following example shows more about it.

Example 2.3. In this example we study the behavior of a boiler device that controls if the pressure variable 'x' is regular or high. There are many ways of designing an algorithm that balances the pressure, at least with respect to a goodness degree. The value of the algorithm is a good reference to decide if it is good enough or not.

Lets consider the fuzzy sets for 'regular pressure' and 'high pressure' defined respectively by $\mu_{rp}, \mu_{hp} : \mathbf{R} \to [0,1]$, where

$$\mu_{rp}(x) = e^{-\frac{1}{2}(x-1.75)^2}$$

$$\mu_{hp}(x) = \begin{cases} 0 & x < 1 \\ \frac{2(x-1)^2}{3} & x \in [1,2] \\ \frac{x}{x+1} & x > 2 \end{cases}$$

The following explanatory data base will be considered here:

$$\text{ED} = \text{PRESSURE[Boiler, Num]}+$$

$$\text{REGULAR[Num; } \mu_{rp}]+$$

$$\text{HIGH[Num; } \mu_{hp}]$$

```
{ $fPre : x \in [0, 7] \wedge x = X_0$}
fassert { $fPre$};deg1; case lab: Tlabel {
                    true, almost true: goto begin
                    default goto error
                    };
begin: { $t = 0$;
{$\alpha_1 = eval( fPre, t := 0, I)$;}
fassert {I};deg2; case lab: Tlabel {
                    true, almost true: goto iter
                    default goto error
                    };
iter: while $High(x)$ do  {;

{$\alpha_3 = eval(I \wedge High(x), x = succ_i(x); t = t + 1, I)$;}
                   $x = succ_i(x)$;
                   $t = t + 1$;
{$\alpha_2 = eval(I \wedge \neg High(x) \Rightarrow fPost)$;}
fassert {$I \wedge \neg High(x) \wedge Reg(x)$}; case lab: Tlabel {
                                    true: goto end
                                    default goto iter
                                    };
 };
 };
end: msg(‘pressure is now regular’);
error: msg(‘It doesn’t work efficiently’)
{$\beta = eval( fPost : Reg(x) \wedge \neg High(x))$}
```

FIGURE 1
Program$_i$.

where the number variable that it is considered is a real number in the universe of discourse

$$\mathbf{U} = [0, 7]$$

Our algorithm will show how to solve the problem of balancing the pressure of a boiler machine: most problems are solved by iteration until the objective is reached, by considering the above iteration fuzzy rule in order to evaluate the algorithm.

The result will depend on a variable of control of the loop and its related successor function.

Figure 2 shows the code of the general algorithm. The successor function should be decided later on.

According to Zadeh’s logic we evaluate all premises where assertions fit in the following way:

$$\{Q\} \equiv \{ fPre\}$$

$$\{R\} \equiv \{ fPost\}$$

(1) Evaluation of premise 1:

$$\alpha_1 = eval(\{fPre\}, t := 0, \{I\}) = 1\ \forall x$$

where invariant condition is the formula

$$I = \{x \in [0,7] \wedge x = succ_i^t(X_0)\}$$

(2) Evaluation of premise 2:

$$\begin{aligned}\alpha_3 &= eval(\{I \wedge \neg High(x) \to fPost\})\\ &= eval(\{High(x) \vee Reg(x)\})\end{aligned}$$

(3) Evaluation of premise 3:

$$\begin{aligned}\alpha_2 &= eval(I \wedge High(x), SInst, I)\\ &= eval(\{\neg High(x) \vee succ(x) \in [0,7]\})\end{aligned}$$

where $SInst \equiv x := succ_i(x); t := t+1$

(4) The antecedent reliability degree is the aggregation

$$\alpha_{123} = \bigwedge_{1 \le i \le 3} \alpha_i$$

this value play the role of point estimation for α_4

(5) Postcondition evaluation: final goal value

$$\begin{aligned}\beta &= eval(\{fPost\})\\ &= eval(\{\neg High(x) \wedge Reg(x)\})\end{aligned}$$

Of course, all these variables take different values depending on the fuzzy framework we were working on.

In order to put our example into practice, lets consider the following cases with respect to the successor function:

1. Successor 1:

$$succ_1(x) = x - c$$

(it will be considered $c = 1$)

2. Successor 2:

$$succ_2(x) = x - \sqrt{x} + 1$$

3. Successor 3:

$$succ_3(x) = \frac{1+x}{2}$$

These cases provide different programs (*Program*$_1$, *Program*$_2$ and *Program*$_2$ respectively) which evaluations help us to decide which one is better.

Tables below show data from some running examples about the input 5.25. The data is obtained by application of evaluation and the inference fuzzy rules on Zadeh's logic. In order to compare more data the execution is forced to continue during seven iterations.

2.4 Case of study 1: successor 1

Iter.	x	High(x)	Reg(x)	α_{123}	β
0	5.25	0.840	0.002	0.840	0.002
1	4.25	0.809	0.043	0.809	0.043
2	3.25	0.764	0.324	0.764	0.235
3	2.25	0.692	0.882	0.882	0.307
4	**1.25**	**0.041**	**0.882**	**0.882**	**0.882**
5	0.25	0	0.324	0.375	0.324
6	−0.75	0	0.043	0.959	0.043
7	−1.75	0	0.002	0.002	0.002

Here there is only one acceptable output: $x = 1.25$. This situation shows the importance of the fuzzy evaluation. Function β increases its value in the first few iterations and then begins to decrease. Just then is when process must be stopped, and recover the last value as the best output, since it reaches the best evaluation for postcondition: $\beta = 0.882$.

2.5 Case of study 2: successor 2

Iter.	x	High(x)	Reg(x)	α_{123}	β
0	5.25	0.840	0.002	0.840	0.002
1	3.95	0.798	0.087	0.798	0.087
2	2.96	0.748	0.475	0.748	0.251
3	2.24	0.691	0.884	0.884	0.308
4	1.74	0.372	0.999	0.999	0.627
5	**1.42**	**0.120**	**0.948**	**0.879**	**0.879**
6	1.23	0.035	0.874	0.874	0.874
7	1.12	0.009	0.820	0.820	0.820

The process stops when the optimal value for β is reached. Note that the 4th iteration reaches the value $x = 1.747$ with $High(1.747) = 0.372$, which is not too much higher. However, it is very important to iterate once again that in order to obtain a better β value: from 0.627 to 0.879, it is worth to run the cycle while β increases.

2.6 Case of study 3: successor 3

Iter.	x	High(x)	Reg(x)	α_{123}	β
0	5.25	0.840	0.002	0.840	0.002
1	3.12	0.757	0.388	0.757	0.242
2	2.06	0.673	0.952	0.952	0.326
3	1.53	0.188	0.976	0.976	0.811
4	**1.26**	**0.047**	**0.889**	**0.889**	**0.889**
5	1.13	0.011	0.826	0.826	0.826
6	1.06	0.002	0.791	0.791	0.791
7	1.03	0.001	0.773	0.773	0.773

In this case something similar happens. The value $x = 1.531$ is not too much higher but a new iteration is considered in order to improve the fuzzy postcondition evaluation.

The use of assert statement join to evaluations in fuzzy logic is then an alternative to improve processes mainly when information contains important fuzzy relations.

3 FUZZY SPECIFICATION OF SYSTEMS

Software specification of requirements involves fuzzy relations as well. Fuzzy sets and fuzzy logic are also powerful tools to increase the quality of the information from software specification. Some technical concepts are explained now in order to understand our proposal to introduce fuzzy logic into formal specification of systems.

3.1 Fuzzy specification in formal methods

Formal specification of algorithms and processes can be improved by introducing fuzzy logic into formal methods. Formal methods are useful to produce documentation free of ambiguity and imprecision in crisp specification of systems. Our main objective is to generalize formal methods into a fuzzy framework. First of all we introduce some concepts in which formal methods are founded.

A formal language **L** is a language defined by precise mathematical formulas. Any language **L** can be viewed as a subset of **A***, where **A*** denotes the set of all words over alphabet **A**. A formal language must be formally specified, for example as strings produced by some formal grammar, strings described by a regular expression, or strings accepted by a finite automata between others. Given a formal language in terms of usual sets, operations between them will produce other formal languages. Concatenation, intersection, union, complement and Kleene star are some of the most useful operations for making new languages. Automata theory in general is a good reference for this topic, see [11].

A specification language is a formal language used during system analysis, requirements analysis and design in a software project. The main goal of a specification language is to describe the system at a much higher level than a programming language, nearer to natural language from where comes all information about the system. Specification language is used for describing 'what' but not 'how', so implementation details are not suitable.

There are two types of specification languages: property-oriented and model-oriented. In the property-oriented approach, specifications of programs consist of logical axioms in a logical system with equality, describing the properties that the functions are required to satisfy. CAST or Common Algebraic Specification Language is a good sample based on first-order logic. On the other hand, model-oriented specifications consist of a realization of the required behavior. *Z notation* is one of this formal specification languages. It is based upon axiomatic set theory, first-order predicate logic and lambda calculus. In any case, specifications must be refinement before be implemented.

Formal specification is a mathematical description of software (or hardware). It may be used to develop implementation of software and it describes what the system should do. Developed design and code will depend on a previous good formal specification of the system. After that, formal verification is used for proving the correctness of algorithms with respect to the formal specification, using formal methods of mathematics, proving theorems concerning properties that specification shows.

Formal methods are based upon mathematics and they can be used to produce documentation in which information is written in a good level of abstraction. Those documents are free of ambiguity and imprecision. However, information is translated in a crisp way before producing documentation. Our goal here is to take information from natural language (in fuzzy terms) and apply formal methods to specify formally the system. *Z notation* [25, 29] is a particular formal method based upon set theory and mathematical logic. This makes *Z notation* the ideal formal method to introduce fuzzy logic in formal specifications.

3.2 Z notation and fuzzy logic

Formal methods as *Z notation*, model the system by means of mathematical objets, functions and relations. In *Z notation* there are several ways for defining an object and several rules for reasoning with the information that they contain. The '*schema signature*' in *Z notation*, defines the entities that make up the state of the system and the schema predicate sets outs conditions that must always be true for these entities [24]. Where a schema defines an operation, the predicate may set out pre- and post-conditions, in this way the state is defined before and after the operation. Here we find a new place where formalizing fuzzy uncertainly. Our next proposal in this paper is to model the information by means of fuzzy logic and fuzzy sets, and formalize the objets and its relations by means of computer with words techniques [31].

As L. A. Zadeh shows in [31], sentences and propositions from natural language must be translated into fuzzy logic formulas (the initial data set), here a normal form it is proposal. Then computing with words and fuzzy inference computes the new states as running the process until the terminal data set is computed. A final step will translate the normal form with computing with words techniques into natural language again. These techniques can be useful for modelling objects and its properties adding a new component to formal specification.

$$\begin{gathered}
\textit{Neighbourhood} \rightarrow \text{poor} + \text{middle} + \text{good} + \text{upper} \\
\textit{Light} \rightarrow \text{dark} + \text{lowshining} + \text{shining} + \text{sunny} \\
\textit{Views} \rightarrow \text{bad} + \text{acceptable} + \text{good} \\
\textit{Rooms} \rightarrow \textit{1} + \textit{2} + \textit{3} + \textit{4} + > \textit{4}
\end{gathered}$$

Obviously, the realtor uses natural language to inform us as an expert that use all his or her previous experiences to make a fast and good valuation.

The following explanatory data base is used for modelling this framework:

$$\begin{aligned}
\textit{ED} = \textit{ESTAT}[\textit{ID}, \textit{List}] + \textit{NEIGHBORHOOD}[X_{neighbourhood}, \mu_{neighbourhood}] \\
+ \textit{LIGHT}[X_{light}, \mu_{light}] + \cdots
\end{aligned}$$

where each crisp or fuzzy characteristic is modeled by a fuzzy set.

Computing with words provides a way of computing mathematical objects that formal methods, as *Z notation*, state to discover solutions and prove that designs fit to the specification. As computing with words, formal methods use the natural language to relate the mathematics to objects in the real world, and computing with words analysis can be very interesting before formalizing into crisp asserts and proof the system.

The goal of *Z notation*, like other formal methods, is to '*add precision to aid understanding*' [29]. *Z notation* is defined in [29] as a mathematical language with a powerful structuring mechanism, that in combination with natural language can be used to produce formal specifications. *Z notation* offers mechanisms that look very adaptable to a fuzzy framework. A collection of linguistic labels can be defined as a 'Free Type'. Free types in *Z notation* are used to model enumerated collections, so we can define

$$\textit{Class}_3 ::= \textit{label} \ll 0..2 \gg$$

and give names to the four elements of the set *Class*:

$$\begin{array}{|c}
\text{bad, regular, good} : \mathit{Class}_3 \\ \hline
\text{bad} = \text{label } 0 \\
\text{regular} = \text{label } 1 \\
\text{good} = \text{label } 2
\end{array}$$

Here, ‘bad’, ‘regular’ and ‘good’ are linguistic labels that we can also assign to a function (μ_{bad}, $\mu_{regular}$, μ_{good}) defined as relations. We can also define relations between them, like ordering:

$$\begin{array}{|c}
\leq_{label} : \mathit{Class}_3 \leftrightarrow \mathit{Class}_3 \\ \hline
\forall l_1, l_2 : \mathit{Class}_3 \bullet l_1 \leq_{label} l_2 \Leftrightarrow \varphi
\end{array}$$

where φ is a first order logic formula or a fuzzy constraint.

Mathematical objects and properties will be collected together in ’*schemas*’. A ‘*scheme*’ in *Z notation* is a pattern of declaration and constraint. It consists of two parts: a declaration of variables, and a predicate constraining their values. A scheme is denoted in the following way:

$$\mathit{Name} \mathrel{\widehat{=}} [\mathit{declaration} | \mathit{predicate}]$$

Every specification in the fuzzy framework can be formalized in *Z notation*. In particular, it is possible to define the types used in the system, by means of *schemes*, as we show in the following example.

Example 3.1. A real estate uses a software system to keep data about estates and make valuations about them. It is easy to understand that coexistence of fuzzy and crisp relations is mandatory because of several factors as brightness, views, size, etc. The object ‘*Estate*’ is defined in general by means of its identification name and a list of characteristics.

$$\mathit{ESTATE} = (\mathit{ID}, [\mathit{Neighbourhood, Light, Views, Rooms, Size}, \ldots])$$

where fuzzy characteristics as ‘*Neighborhood*’, ‘*Light*’ and ‘*Views*’ are fuzzy granules in the sense of [31], and there are also crisp characteristics like number of rooms. Taking into account that fuzzy characteristics are going to be computed by means of fuzzy sets, we can formalize the following types in *Z notation*:

- *Estate* is a record that contains the following characteristics in our problem:

$$\mathit{Estate} \mathrel{\widehat{=}} [\mathit{Id}, \mathit{rooms} : \mathbb{N}; \mathit{size} : \mathbb{R}; \mathit{light}, \mathit{views} : \mathit{Granule} | \mathit{size} > 0]$$

- *Granule* defines a fuzzy granule in the sense of [31], with the following formal specification in *Z notation*:

$$\mathit{Granule} \mathrel{\widehat{=}} [k : \mathbb{N}; \mathit{label} : \mathit{LingLable}[0 \ldots k] |$$
$$(kmod2 = 0) \wedge (\forall i \in \{0..k\} \exists \mu_i : Trapezoidal \bullet label_i \leftrightarrow \mu_i) \wedge$$
$$(\forall i \in \{0..k-1\} label_i \prec label_{i+1})]$$

- *Trapezoidal* defines a fuzzy set by means of a μ function, formalized in the following way:

$$\mathit{Trapezoidal} \mathrel{\widehat{=}} [\mu : \mathbb{R} \mapsto [0,1]; a, b, c, d : \mathbb{R} |$$
$$(0 \le a < b \le c < d) \wedge \mu(x) = \begin{cases} 0 & (x < a) \vee (x > d) \\ \frac{x-a}{b-a} & x \in [a,b] \\ 1 & x \in [b,c] \\ \frac{d-x}{d-c} & x \in [c,d] \end{cases}$$

- The subtype *Triangular* of *Trapezoidal*, is a useful type:

$$\mathit{Triangular} \subseteq \mathit{Trapezoidal} | c = d$$

Z notation provides also schemas for declarations, predicates or renaming. The information contained in schemas may be combined by conjunction, disjunction, negation, quantification and composition. This application of the schema language allows to represents a state of the system as an object of the corresponding schema type, and make reasoning about computing states during a process or an algorithm.

4 CONCLUSIONS AND FUTURE WORK

We want to stress again how relevant developing fuzzy specification of algorithms and processes will be in the next future, by introducing computer with words techniques into formal methods. It will be extremely relevant in early stages of the process, when the specification is 'customer-oriented' and natural language is used. In particular, in this paper we have presented a first proposal on how specification of algorithms, together with pre and post-conditions, can be improved if fuzzy relations and constraints are allowed. Our objective is to allow software specification process information given in natural language, which implies a lot of fuzzy concepts and perceptions. Granulation and computing with words, see [31], take into account the existence of fuzzyness within specification processes, so we can avoid lose of information due to lack of accuracy of alternative crisp approaches. In addition, the model should get closer to non-technical users.

We must of course acknowledge the amount of work ahead, which obviously includes rigorous demonstrations based upon experimentation [28], with real data or simulation (see, e.g., [13, 22, 23]). Examples along the text have been introduced to illustrate ideas. The key issue it to realize that formal fuzzy specification is a necessary tool for the improvement of the system design and any required process or algorithm associated to a project subject to linguistic input information. The relevance of these studies is made more clear as soon as we realize that mathematical models should rarely cross the decision-aid role, and the difficulties most non-technical decision makers will find in understanding non-linguistic output information [19].

ACKNOWLEDGEMENTS

This research has been partially supported by the Government of Spain, grant number TIN2006-01960. we also want to thank anonymous referees for their constructive suggestions, which indeed improved the paper and will be very helpful in future research. Some of their comments have been included in the text.

REFERENCES

[1] J. M. Abietar, P. J. Morcillo and G. Moreno. Building a fuzzy logic programming tool, *Proceedings of the Spanish Conference on Informatics, CEDI*, Thomsom-Paraninfo, Zaragoza, 2007, pp. 215–222.

[2] B. Jose. *Assertions in Java*, http://javaboutique.internet.com/tutorials/assertions.

[3] R. Bird. *Introduction to functional programming using Haskell*, Prentice Hall, 1998.

[4] T. Calvo, A. Kolesarova, M. Komornikova and R. Mesiar. Agreggation operators: properties, classes and construction methods. In T. Calvo, G. Mayor and R. Mesiar (Eds.), *Aggregation Operators*, Physica-Verlag, Heidelberg, 2002, pp. 3–104.

[5] T. Calvo and A. Pradera. Double aggregation operators. *Fuzzy Sets and Systems* **142** (2004), 15–33.

[6] E. W. Dijkstra and W. H. J. Feijen. *A method of programming*, Addison-Wesley, 1988.

[7] A. Gray and S. MacDonell. Fuzzy logic for software metric models throughout the development life-cycle, *Proceedings of the North American Fuzzy Information Society conference*, New York, 10–12 June, 1999, pp. 258–262.

[8] A. Hall and R. Chapman. Correctness by construction: developing a commercially secure system. *IEEE Software* **19** (2002), 18–25.

[9] A. Hall. Realising the Benefits of Formal Methods. *Journal of Universal Computer Science* **13** (2007), 669–675.

[10] C. A. R. Hoare. An axiomatic basis for computer programming, *Communications ACM* **12** (1969), 89–100.

[11] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory and Computation*, Addison-Wesley, 1979.

[12] A. Idri, A. Abran and T. M. Khoshgoftaar. Computational intelligence in empirical software engineering, *Proceedings of the First USA-Morocco Workshop on Information Technology*, Rabat, Morocco, 2003.

[13] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard. P. W. Jones, D. C. Hoaglin, K. E. Emam and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transaction on Software Engineering* **28** (2002), 721–734.

[14] V. López, J. M. Cleva and J. Montero. A functional tool for Fuzzy First Order Logic Evaluation. In D. Ruan, P. D'hont, P. F. Fantoni, M. De Cock, M. Nachtegael and E. E. Kerre, (Eds.), *Applied Artificial Intelligence*, World Scientific, New Jersey, 2006, pp. 19–26.

[15] V. López and J. Montero. Fuzzy specification of algorithms, *Proceedings of the New Trends in Preference Modeling, Eurofuse Workshop*, A Demanda, Jaén, 2007, pp. 151–156.

[16] V. López, J. Montero. Fuzzy specification in software engineering, *Proceedings of the Joint Conference on Infomation Sciences*, World Scientific, Salt Lake City, 2007.

[17] V. López, J. Montero, L. Garmendia and G. Resconi. Specification and computing states in fuzzy algorithms. *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems*, to appear.

[18] B. Meyer. *Eiffel: The language*, http://www.engin.umd.umich.edu/CIS/course.des/cis400/eiffel/eiffel.html, 1992.

[19] J. Montero, V. López and D. Gómez. The role of fuzziness in decision making. In D. Ruan *et al.*, (Eds.), *Fuzzy Logic: an spectrum of applied and theoretical issues*, Springer, 2008, pp. 337–349.

[20] J. Montero and M. Mendel. Crisp acts, fuzzy decisions. In S. Barro *et al.*, (Eds.), *Advances in Fuzzy Logic*, University of Santiago de Compostela, 1998, pp. 219–238.

[21] W. Pedrycz. Computational intelligence as an emerging paradigm of software engineering, *Proceedings of the International conference on Software engineering and Knowledge Engineering*, ACM International Conference Proceedings, 2002, pp. 7–14.

[22] C. B. Seaman. Qualitative methods in empirical studies of software engineering, *IEEE Transaction on Software Engineering* **25** (1999), 557–572.

[23] D. I. K. Sjoberg, B. Anda, E. Arisholm, T. Dyba, M. Jorgensen, A. Karahasanovic, E. F. Koren and M. Vokác. Conducting realistic experiments in software experiments, *Proceedings of the International Symposium on Empirical Software Engineering, ISESE*, IEEE Press, 2002, pp. 17–26.

[24] I. Sommerville. *Software Engineering*, Addison-Wesley, 2004.

[25] J. M. Spivey. *The Z notation: A Reference Manual*, Prentice-Hall, 1998.

[26] Standish Group. *The Standish Group Chaos Report*, http://www.projectsmart.co.uk/docs/chaos_report.pdf, 1995.

[27] R. Thayer and M. Dorfman. *Software Requirements Engineering*, IEEE Computer Society Press, 1997.

[28] F. Tichy. Should computer scientists experiment more? *Computer* **31** (1998), 32–40.

[29] J. Woodcock and J. Davis. *Using Z: Specification, Refinement and Proof*, Prentice Hall, 1997.

[30] XFuzzy 3.0. National Center of Microelectronics. Microelectronics Institute, Sevilla (Spain). http://www.imse.cnm.es/xfuzzy.

[31] L. A. Zadeh. From computing with numbers to computing with words – From manipulation of measurements to manipulation of perceptions. In P. P. Wang, (Ed.), *Computing with words*, Wiley, New York, 2001, pp. 35–68.