Information and Software Technology 43 (2001) 725–741

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Leaving inconsistency using fuzzy logic

Francesco Marcelloni[a,*], Mehmet Aksit[b]

[a]Dipartimento di Ingegneria della Informazione, Elettronica; Informatica, Telecomunicazioni, University of Pisa, Via Diotisalvi, 2-56122 Pisa, Italy
[b]Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

Received 22 June 1999; revised 8 August 2001; accepted 14 August 2001

Abstract

Current software development methods do not provide adequate means to model inconsistencies and therefore force software engineers to resolve inconsistencies whenever they are detected. Certain kinds of inconsistencies, however, are desirable and should be maintained as long as possible. For instance, when multiple conflicting solutions exist for the same problem, each solution should be preserved to allow further refinements along the development process. An early resolution of inconsistencies may result in loss of information and excessive restriction of the design space. This paper aims to enhance the current methods by modeling and controlling the desired inconsistencies through the application of fuzzy logic-based techniques. It is shown that the proposed approach increases the adaptability and reusability of design models. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Inconsistency resolution; Software development methods; Fuzzy logic; Adaptable design models; Software artifacts

1. Introduction

Due to size and complexity of today's applications, developing cost-effective software systems is a difficult task. Further, software engineers generally have to deal with various kinds of inconsistencies that may originate from requirement specifications, involvement of multiple persons in the same project, errors in the software development process, alternative solutions, etc. [14]. Certain kinds of inconsistencies are inevitable, for instance, in case multiple persons working independently of each other within the same project [30]. Some inconsistencies are desirable when, for instance, alternative solutions exist for the same problem, and these solutions have to be preserved to allow further refinement along the development process [27]. In particular, alternative solutions manifest themselves in software systems affected by continuously changing requirements. This paper focuses on modeling and handling desirable inconsistencies.

During the last decade, a considerable number of software development methods have been introduced [26,31]. Methods aim to create software artifacts through the application of a number of rules. For example, the OMT method [26] introduces rules for identifying and discarding object-oriented artifacts such as classes, associations, and part-of and inheritance relations. These methods do not define means to model the desired inconsistencies and, therefore, aim to resolve inconsistencies whenever they are detected. For example, in object-oriented methods a candidate class is generally identified by applying the rule: If an entity in a requirement specification is relevant and can exist autonomously in the application domain then select it as a candidate class. While applying the object-oriented intuition of what a class should be, this rule follows the consistency constraint 'an entity is either a candidate class or not a candidate class but not partially both'. In this example, the software engineer has to determine whether the entity being considered is relevant or not for the application domain. The software engineer can perceive that the entity partially fulfils the relevance criterion and may conclude that the entity is, for instance, substantially relevant. This definition would imply the classification of the entity as a partial class, which is considered as an inconsistent class definition by the current object-oriented methods. Therefore, the consistency constraints force the software engineer to take abrupt decisions, such as accepting or rejecting the entity as a class. This results in loss of information because the information about the partial relevance of the entity is not modeled and therefore cannot be considered explicitly in the subsequent phases [22].

In this paper, we propose a fuzzy logic-based software development technique for coping with inconsistencies.

* Corresponding author. Tel.: +39-50-568-678; fax: +39-50-568-522.
E-mail addresses: f.marcelloni@iet.unipi.it (F. Marcelloni), aksit@cs.utwente.nl (M. Aksit).

This technique increases the adaptability and reusability of design models and is not specific to a particular method. Fuzzy logic can express uncertainty and imprecision. Further, fuzzy logic provides a sound framework to define a language, to associate a meaning with each expression of the language and to compute these expressions. These features make fuzzy logic more advantageous than other approaches in representing naturally qualitative aspects of data and heuristic rules generated from the experience [8,33]. A software engineer can describe his/her perception using his/her natural language and this perception can be modeled and maintained along the overall development process. Thus, an entity can, for instance, be considered both as a weak class and as a substantial attribute. Although class and attribute are two conflicting design alternatives for an entity, fuzzy logic allows managing this inconsistency. The linguistic expressions used to qualify the object-oriented concepts (weak and substantial in the previous example) can be considered as measures of each alternative. These measures prove to be particularly useful in selecting the best alternative in a set of possible conflicting design alternatives. Capturing as much as possible the perception of the software engineer reduces the loss of information and, consequently, improves the quality of the software development process.

2. Early resolution of inconsistency

2.1. Methodological rules

Methods create software artifacts by exploiting the underlying concepts[1] through the application of heuristic rules. Each rule is derived from a particular intuition of the artifact being developed. For instance, a candidate class is identified as an entity, which is relevant and can exist autonomously in the application domain. Further, the intuition of the artifact candidate class expresses that the more an entity is relevant and can exist autonomously, the more the entity matches the concept of candidate class. The intuition, therefore, involves a gradation of matching. With respect to relevance and autonomy in the application domain, an entity can be considered as a partial candidate class. From the software engineer's perspective, a partial classification means that there is not sufficient evidence to take an abrupt decision such as to accept or reject the entity as a class. Relevance and autonomy are only a view of the concept of class: each view expresses its opinion and this opinion can reinforce a design alternative (the entity is a class) with respect to the other (the entity is not a class), or vice versa. Only when all the relevant opinions are collected, the entity can be accepted or rejected as a class. In the mean time, the possible conflicting design alternatives have to be maintained.

[1] For example, object-oriented methods exploit the concepts like object, class, aggregation and inheritance.

On the contrary, current methodological rules impose to each view to express not an opinion, but an abrupt classification. Consider, for instance, the following rule *Candidate Class Identification* used by some popular object-oriented methods[2] to identify candidate classes:

> **IF** AN ENTITY IN A REQUIREMENT SPECIFICATION IS RELEVANT **AND** CAN EXIST AUTONOMOUSLY IN THE APPLICATION DOMAIN **THEN** SELECT IT AS A CANDIDATE CLASS.

Here, *an entity in a requirement specification* and *a candidate class* are the two object-oriented artifact types to be reasoned. If the antecedent of the rule is *true*, then the result of this rule is the classification of an entity in a requirement specification as a candidate class. This rule does not consider gradation in perception: an entity is either a candidate class or not. It follows that a unique view of the concept of class completely decides whether an entity can be accepted or rejected as a class. Once an artifact has been classified, for instance, into the rejected set of a rule, it is not considered anymore by the rules that apply to the accepted set of that rule. Of course, a rejected entity can be considered by another rule, which applies to the entities in a requirement specification. Consider, for example, rule *Candidate Attribute Identification*:

> **IF** AN ENTITY IN A REQUIREMENT SPECIFICATION IS RELEVANT **AND** CANNOT EXIST AUTONOMOUSLY IN THE APPLICATION DOMAIN, **THEN** IDENTIFY IT AS A CANDIDATE ATTRIBUTE

.

This rule can be applied to the entities in a requirement specification, which are rejected by rule *Candidate Class Identification*. If all the rules, which are applicable to an entity in a requirement specification, reject that entity, then the entity is practically discarded. Here, the rule follows the consistency constraint imposed by the artifact type Candidate Attribute: 'an entity is either a candidate attribute or not'. Further, it is obvious that if an entity is a candidate class, it cannot be a candidate attribute and vice versa.

Enforcing consistency constraints do not allow software engineers to reason on different design solutions concurrently. New properties corresponding to different views of concepts can only cause the conversion from a design solution to a conflicting design solution. For instance, when investigating the functional view, an attribute can be converted to a class by rule *Attribute to Class Conversion*:

> **IF** A CANDIDATE ATTRIBUTE IS RESPONSIBLE FOR THE REALIZATION OF FUNCTIONS **THEN** CONVERT THE CANDIDATE ATTRIBUTE TO CLASS.

[2] The approach proposed in this paper is not restricted to object-oriented methods. We adopted an object-oriented method because of our background.

Similarly, a class may be converted to an attribute by rule *Class to Attribute Conversion*.

IF A CANDIDATE CLASS IS NOT RESPONSIBLE FOR THE REALIZATION OF ANY FUNCTION **THEN** CONVERT THE CANDIDATE CLASS TO ATTRIBUTE.

In case candidate classes and attributes are not converted, they are selected as classes and attributes, respectively. After identifying classes, inheritance and aggregation relations are determined, for example, based on the following rules:

Aggregation Identification

IF CLASS A CONTAINS CLASS B, **THEN** CLASS A AGGREGATES CLASS B.

Inheritance Identification:

IF CLASS A IS A KIND OF CLASS B, **THEN** CLASS A INHERITS FROM CLASS B.

Despite reduced number of rules, the example method shown in this section highlights how methodological rules are chained with each other, that is, the output of a rule is input to another rule. This implies that bad decisions taken in the first levels of the rule chain have repercussions on the subsequent levels. For instance, when identifying inheritance or aggregation relations, if entities have been misclassified as classes or as non-classes, inheritance and aggregation relations will not be defined correctly in their turn.

In the later phases of the development process, when the final structure of the software is almost defined, heuristics can be based on more precise and objective inputs than in the first phases. The application of these heuristics can validate the design choices or trigger a reevaluation of these choices. For instance, class hierarchy can be modified using the following rule *Inheritance Modification*:

IN THE CLASS HIERARCHY, **IF** THE NUMBER OF IMMEDIATE SUBCLASSES SUBORDINATED TO A CLASS IS LARGER THAN 5, **THEN** THE INHERITANCE HIERARCHY IS COMPLEX.

This rule is extracted by the metrics proposed in Ref. [7]. If this rule concludes that the inheritance hierarchy is complex, then the hierarchy may be modified.

For the sake of brevity, in this section we have introduced only a subset of the heuristics composing a method. We would like to point out, however, that the solutions proposed in the next sections can easily be extended to a complete set of heuristics.

2.2. Quantization error

The consistency constraints enforced in current methods impose abrupt classifications. For instance, rule *Candidate Class Identification* asks a software engineer to classify the entities as relevant or not relevant entities. Although the software engineer may perceive that an entity partially fulfils the relevance criterion, the rule imposes her/him to take an abrupt decision: to accept or reject the entity as a candidate class. In general, therefore, application of a rule *quantizes* a set of object-oriented artifacts into two subsets: *accepted* or *rejected*. If on the one side this strategy reduces the complexity of the design process, on the other side it generates the so-called quantization error [22].

To make the concept of quantization error clear we can refer to the area of digital signal processing. Here, quantization process consists of assigning the amplitudes of a sampled analog signal to a prescribed number of discrete *quantization levels*. This results in loss of information because the quantized signal is an approximation of the analog signal. Quantization error is defined as the difference between an analog and the corresponding quantized signal sample. The less the number of quantization levels, the higher the quantization error.

To enforce consistency constraints, in current methodological rules, high quantization errors arise from the fact that rules adopt only two quantization levels. Here, the quantization error is the difference between the perception of the software engineer and the two 'quantization levels' imposed by methodological rules.

One of the dramatic effects of the quantization error on the development process is early elimination of artifacts. Each decision taken by a rule is based on the available information up to that phase. For the early phases, there may not be sufficient amount of information available to take abrupt decisions like discarding an entity. Such an abrupt decision must be taken only if there is sufficient evidence that the entity is indeed irrelevant. In most object-oriented methods, however, each identification process is followed by an elimination process. For example, the OMT method [26] proposes a process that includes class identification and elimination, association identification and elimination, and so on. Now, assume that a software engineer discards an entity because it is considered non-relevant. The discarded entity, however, could have been included as a class, if the software engineer had gathered more information about its structure and operations. During the later phases, this would be practically impossible because the discarded entity could not be considered further. Early elimination of artifacts in current methods is practically inevitable.

If, at the end of the development process, the software engineer realizes that the resulting object model is not satisfactory, there are two possible options: improving the model by applying subsequent rules and/or by iterating the process. The application of subsequent rules may not adequately improve the model because of the loss of information due to quantization errors. The iteration of the process still suffers from the quantization error problem. Moreover, managing iterations remains as a difficult task.

3. Requirements for improving current methods

We propose the following requirements for improving current methods:

- *Reducing the quantization error*: To reduce the quantization error and its negative effects, desired inconsistencies should be preserved and resolved only when it is necessary. Such an inconsistency resolution, for instance, may be requested by the language compiler. A demand for resolving an inconsistency therefore may be context or language dependent. For example, the C++ language allows multiple inheritance specification whereas the Smalltalk language forbids it. The objective of preserving inconsistencies, however, has not to be achieved to the detriment of the intuitiveness of the methods. Indeed, software development process is a highly labor intensive work and therefore the adopted rules, alternatives and measures must be expressed in an intuitive way. Preferably, well-established rules and heuristics of the current methods must be respected.
- *Provide a measure for alternatives*: Preserving alternative solutions does not mean that all the alternatives are equally valid. Consider for example that an entity may be classified as an attribute and a class at the same time. To be able to reason about the alternatives, there is a need to give a measure for each alternative. The software engineer, for instance, may classify an entity more like a class than an attribute and give a higher measure to it.
- *Manage complexity*: Deferring consistency enforcement decreases the loss of information but increases the complexity of design. There is a need for introducing appropriate techniques to manage this increased complexity without necessarily giving up the design flexibility. In particular, the trade-off between flexibility and complexity should be controlled by the software engineer.

4. Using fuzzy logic in modeling inconsistencies

4.1. Modeling artifacts

Denote each artifact type as $[T, (P_1, D_1), (P_2, D_2), \ldots, (P_n, D_n)]$, where T is the artifact type name, P_i is a property of T and D_i is the definition domain of P_i. An example of an artifact type is [*Entity*,(*Relevance*,{*True*, *False*}), (*Autonomy*,{*True*, *False*})]. Here, *True* and *False* are the only two values that Relevance and Autonomy can assume in current methodological rules. A software artifact is an instantiation of its type and can be expressed as $Name \leftarrow [T, (P_1 : V_1), (P_2 : V_2), \ldots, (P_n : V_n)]$, where T is the artifact type, *Name* is the name of the artifact, and V_i is a value defined on domain D_i of property P_i. In the following example, *Square* is an instance of artifact type Entity:

$$Square \leftarrow [\textbf{\textit{Entity}}, (\textbf{\textit{Relevance}} : True), (\textbf{\textit{Autonomy}} : True)]$$

Depending on the values of the properties, an artifact can 'evolve' and become an instance of another artifact type. For instance, if the values of properties Relevance and Autonomy are set to True, artifact Square becomes an instance of Candidate Class. Let us denote an artifact type A whose properties are used to determine the set of instances of an artifact type B as pre-artifact type of B. In our example, Entity is a pre-artifact type of Candidate Class. An artifact type can have one or more pre-artifact types. For instance, Candidate Class and Candidate Attribute are both pre-artifact types of Class (see rule *Attribute to Class Conversion*).

The transformation process from instance of an artifact type into instance of another artifact type is controlled by the methodological rules. For instance, rule *Candidate Class Identification* defines the transformation from Entity into Candidate Class. Rules can be expressed using the notation introduced to represent artifacts in the following way:

$$P \leftarrow [\textbf{\textit{Entity}}, (\textbf{\textit{Relevance}} : True), (\textbf{\textit{Autonomy}} : True)]$$

$$\Rightarrow P \leftarrow [\textbf{\textit{CandidateClass}}]$$

Here, P indicates a variable, which is instantiated to the artifact being reasoned on, and $\Rightarrow$ represents the classical implication operator.

The group of rules, which have an artifact type in their consequent part, determines the set of instances of that artifact type. The set of instances x of an artifact type T can be defined as $\{x: C_T(x)\ holds\}$, where $C_T(x)$ are the conditions an artifact has to satisfy to be a member of the set. Conditions $C_T(x)$ correspond to the antecedents of rules which have T in their consequent part, and are typically expressed as logical expressions of properties of one or more pre-artifact types of T. For instance, the set of instances cc of Candidate Class is $\{cc : cc \leftarrow [\textbf{\textit{Entity}}, (\textbf{\textit{Relevance}} : True), (\textbf{\textit{Autonomy}} : True)]\}$. Similarly, the set of instances a of Candidate Attribute is $\{a : a \leftarrow [\textbf{\textit{Entity}}, (\textbf{\textit{Relevance}} : True), (\textbf{\textit{Autonomy}} : False)]\}$.

We define two artifact types A and B to be conflicting if there exists no artifact which can be an instance of both. Let $\{x: C_A(x)\ holds\}$ and $\{x: C_B(x)\ holds\}$ be the sets of instances of artifact types A and B. Then, A and B are conflicting if there exists no artifact x such that $C_A(x)\ \&\ C_B(x) = true$, where & stands for the logical *and*.

From this definition, it can be deduced that artifact types Candidate Class and Candidate Attribute, and Class and Attribute are conflicting artifact types. Further, an artifact cannot be an instance and, at the same time, a non-instance of an artifact type. The conflict is caused from the conditions $C_T(x)$, which trace out abrupt boundaries between instances and non-instances of an artifact type. When method developers define rules, however, they intuit, for instance,

Table 1
Some popular triangular norms and corresponding conorms

T	$T(a,b)$	$T^*(a,b)$
Minimum	$\min(a,b)$	$\max(a,b)$
Product	ab	$a+b-ab$
Bounded product	$\max(0,a+b-1)$	$\min(1,a+b)$
Drastic product	$\begin{cases}\min(a,b) & \text{if } \max(a,b)=1\\ 0 & \text{otherwise}\end{cases}$	$\begin{cases}\max(a,b) & \text{if } \min(a,b)=0\\ 1 & \text{otherwise}\end{cases}$

that entities can be partially relevant and autonomous, and a partially relevant and autonomous entity should be selected as a partial member of both Candidate Class and Candidate Attribute. Nevertheless, they are forced to quantize their intuition of partial relevance and autonomy so as to create a sharp boundary between instances and non-instances of an artifact type. There exists a semantic gap between method developers' intuition of an artifact type and actual representation of this intuition by means of two-valued logic-based rules.

Also, the abrupt boundaries traced out by the conditions do not allow capturing completely software engineers' perception of an artifact. Software engineers, for instance, can perceive different grades of relevance of an entity in their turn, but they are required to quantize their perception in order to match input values permitted by rule *Candidate Class Identification*. There exists a semantic gap between the software engineers' perception and the input required by the rule.

As observed in Lakoff [18], some sets appear to be *graded sets*, that is, they show a fuzzy boundary whose 'width' is defined by a linear scale of values between 0 and 1, with 1 at the interior and 0 at the exterior. Artifact types seem to identify graded sets rather than classical sets [23]: a development process element such as an entity in a requirement specification can be an instance of an artifact type at a certain degree. To reason on graded sets, an appropriate logic has to be used. If n is the number of membership degrees, extensions of two-valued logic to n-valued logics are needed to manage the possible n truth levels. When the number of degrees tends to infinity, graded sets become fuzzy sets and n-valued logic degenerates to an infinite-valued logic. The term infinite-valued logic is usually used in the literature to indicate a logic whose truth-values are represented by all the real numbers in the interval [0,1]. Infinite-valued logic is isomorphic to fuzzy set theory in the same way as the two-valued logic is isomorphic to the crisp set theory [17]. The isomorphism follows from the fact that the logic operations have the same mathematical form as the corresponding operations on fuzzy sets. As the set of fuzzy operations can be implemented in different ways, a variety of fuzzy set theories and therefore of infinite logics (or fuzzy logics) can be derived. In Section 4.2, we will introduce some basic aspects of fuzzy set and fuzzy logic theories to allow readers non-familiar with these concepts to easily comprehend the approach presented in Section 4.3.

4.2. Fuzzy logic

A fuzzy set S of a universe of discourse U is characterized by a membership function which associates with each element u of U a number in the interval [0,1] which represents the grade of membership of u in S [32].

Several operations are defined on fuzzy sets. Given two fuzzy sets A and B in a universe of discourse U, some basic operations are the following:

- Complement $\neg A = \int_U (1-\mu_A(u))/u$;
- Intersection $A \cap B = \int_U T(\mu_A(u), \mu_B(u))/u$;
- Union $A \cup B = \int_U T^*(\mu_A(u), \mu_B(u))/u$.

where the integral sign $\int_U \mu(u)/u$ stands for the union of the points u at which $\mu(u)$ is positive and T and T^* identify a triangular norm and the corresponding conorm, respectively. The definitions of some popular triangular norms and corresponding conorms are given in Table 1.

Relations can be defined among fuzzy sets. A fuzzy relation $R(x_1,\dots,x_N)$ is a fuzzy subset of the Cartesian product $U_1 \times \cdots \times U_N$, where $U_1 \times \cdots \times U_N$ is the collection of ordered tuples $u_1,\dots,u_N$, with $u_1 \in U_1,\dots,u_N \in U_N$. R is characterized by a multivariate membership function $\mu_R(u_1,\dots,u_N)$ and is expressed as:

$$\int_{U_1 \times \cdots \times U_N} \mu_R(u_1,\dots,u_N)/u_1\dots u_N \tag{1}$$

Finally, relations can be composed. Let R and S be fuzzy relations in $U \times V$ and $V \times W$, respectively. The *composition* of R and S is a fuzzy relation denoted by $R \circ S = \int_{U\times W} \sup_{v\in V} T(\mu_R(u,v), \mu_S(v,w))/(u,w)$. R or S can be unary relations. The *sup-T* operator is called *composition operator*.

Based on the definition of fuzzy set, Zadeh introduced the concept of linguistic variable [32]. A *linguistic variable* is a variable whose values, called *linguistic values*, have the form of phrases or sentences in a natural or artificial language. For instance, possible linguistic values of linguistic variable Temperature might be *low*, *medium* and *high*. The meaning of a linguistic value v is the fuzzy set $M(v)$ of universe U for which v serves as label. For instance, the three fuzzy sets shown in Fig. 1 express the meaning of *low*, *medium* and *high temperature*. Here, universe U is defined on the scale of degrees centigrade between 0 and

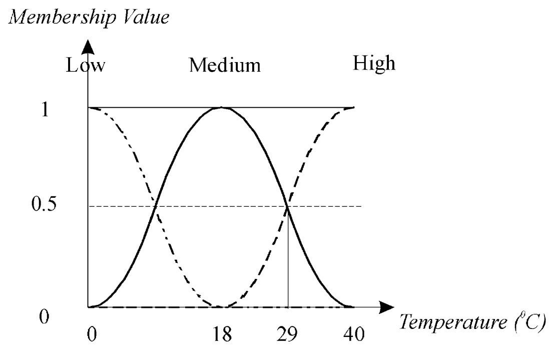


Fig. 1. Linguistic variable Temperature.

40. Each degree of the scale belongs to the fuzzy set associated with a linguistic value at a different grade. For instance, the temperature 29°C belongs to *medium* and *high* with membership value 0.5 and to *low* with membership value 0.

Syntactically, a linguistic value is a composition of the following atomic terms:[3]

1. *Primary terms*, which are labels of specified fuzzy sets in the universe of discourse. For instance, *low*, *medium* and *high* for the linguistic variable Temperature;
2. Negation *not* and connectives *and* and *or*;
3. Markers such as parentheses.

All possible values of a linguistic variable can be generated by a context-free grammar $G = (T, N, P)$, where T and N are the terminal and non-terminal symbols, respectively, and P is the production system. The terminal symbols are the atomic terms. The meaning associated with each possible linguistic value is determined by a semantic rule R, which maps each linguistic expression into an operation on fuzzy sets. For instance, negation *not* complements the corresponding fuzzy set, connectives *and* and *or* are defined as the intersection and the union between fuzzy sets, respectively. The markers change the normal sequence of the operations.

It follows that a linguistic variable is characterized by a quintuple (x, $TN(x)$, U, G, R) where x is the name of the variable, $TN(x)$ is the term set of x, that is, the union of terminal and non terminal symbols of x with each value being a fuzzy set defined on universe U, G is the context free grammar for generating the symbols of x, and R is the semantic rule. The definition of G and R is shared among all the linguistic variables except for the primary terms and their meanings. In general, therefore, a linguistic variable is completely characterized by defining the universe, the labels representing the primary terms and their meanings.

[3] For the sake of simplicity, we do not consider modifiers in the atomic terms. Modifiers are linguistic expressions such as more or less, very, minus, plus, which modify the meaning of the atomic term which they are applied to.

Linguistic variables allow expressing rules in a natural way and the meaning associated with each linguistic value permits to reason on these rules. A fuzzy rule is typically expressed as **IF** X_1 is A_1 **AND**…**AND** X_N is A_N **THEN** Y is B, or for short $I(A_1 \wedge \ldots \wedge A_N, B)$, where X_i, with $i = 1\ldots N$, and Y are linguistic variables defined on the universes U_i and V, respectively, A_i and B are linguistic values of X_i and V, respectively, and I is a fuzzy implication operator. The connective **AND** and the fuzzy implication are implemented as fuzzy relations. For the connective **AND**, $A_1 \wedge \ldots \wedge A_N = \int_{U_1 \times \cdots \times U_N} T(\mu_{A_1}, \ldots, \mu_{A_N})/u_1\ldots u_N$, with T a triangular norm and μ_{A_i} the membership function associated with the primary term A_i. A *fuzzy implication* is defined for all $t \in T$ and $v \in V$ by $I(A, B) = \int_{T \times V} F(\mu_A(t), \mu_B(v))/(t, v)$, where F may be each function from $[0, 1] \times [0, 1]$ to $[0, 1]$ that satisfies the boundary conditions $F(0, 0) = F(0, 1) = F(1, 1) = 1$ and $F(1, 0) = 0$. Several families of fuzzy implication operators have been proposed in literature. Comparative studies can be found in Ref. [17].

Given a fuzzy rule **IF** X_1 is A_1 **AND**…**AND** X_N is A_N **THEN** Y is B and a fact X_1 is $\hat{A}_1$ **AND**…**AND** X_N is $\hat{A}_N$, the inference mechanism used to infer a conclusion B' is normally implemented by a generalization of the modus ponens, called generalized modus ponens or compositional rule of inference. Conclusion B' is computed as $B' = (A'_1 \wedge \ldots \wedge A'_N) \circ I(A_1 \wedge \ldots \wedge A_N, B)$, where $\circ$ denotes the composition operator and I a fuzzy implication operator [32]. The conclusion B' is therefore obtained by first computing the fuzzy sets corresponding to the fact and to the rules, and then composing these fuzzy sets by the composition operator. It follows that in fuzzy logic a reasoning tool like the generalized modus ponens is implemented as a sequence of operations on fuzzy sets. Notice that the generalized modus ponens allows inferring conclusions also if the fact corresponds only approximately to that expected in the antecedent part of the rule.

As noted in Ref. [29], the causal relationship between the variables, which define a real system, cannot be expressed by only one rule. So, typically we should deal with a set of rules: the conclusion inferred from all the rules will be obtained by aggregating the conclusions inferred by the single rules. Aggregation is generally implemented as a fuzzy intersection or union. The choice of the type of aggregation operation depends on the type of fuzzy implication and composition operators. In general, the criterion adopted in the choice is the fundamental requirement for fuzzy reasoning, i.e. given a fact that matches the antecedent of a rule, the conclusion has to match the consequent part of that rule. A detailed analysis on the relationships among types of aggregation, implication and composition operators, and partitions of the input and output spaces for satisfying the fundamental requirement for fuzzy reasoning can be found in Ref. [19].

A conclusion is expressed as a fuzzy set. If we are interested in a crisp value, we can defuzzify the conclusion by a defuzzification strategy [32]. A defuzzification strategy is aimed at producing the crisp value that best represents the linguistic value. At present, the commonly used strategies may be described as the *mean of maxima* and the *center of area*. The crisp value produced by the mean of maxima strategy represents the mean value of the elements, which belong to the fuzzy set characterizing the conclusion with maximum grade. The center of area strategy produces the center of gravity of the fuzzy set characterising the conclusion.

4.3. Fuzzy artifacts

Artifacts can be instances of an artifact type at different grades. To take this membership gradation into account the notation of artifact type introduced in Section 4.1 has to be changed into $[T, (\mathit{Membership}, D_M), (P_1, D_1), (P_2, D_2), \ldots, (P_n, D_n)]$, where T is the artifact type name, Membership contains the membership value, D_M is the definition domain of Membership, P_i is a property of T and D_i is the definition domain of P_i. A software artifact can be expressed as $\mathit{Name} \leftarrow [T, (\mathit{Membership}: V_M), (P_1 : V_1), (P_2 : V_2), \ldots, (P_n : V_n)]$. The membership values depend on the truth-values of antecedents of rules, which have the artifact type in their consequent part. For instance, the membership value of entity to the set of instances of artifact type Candidate Class depends on the values of Relevance and Autonomy.

Let us assume Relevance and Autonomy vary in the interval [0,1]. As the heuristic corresponding to rule *Candidate Class Identification* suggests that the more an entity is relevant and autonomous, the more the entity is an instance of Candidate Class, we could define the membership value of the entity to Candidate Class as the product of relevance and autonomy values. Though this definition is logically correct, it requires a software engineer to input numerical values for properties that cannot be easily quantified. For instance, with respect to Relevance, a software engineer can only express a qualitative evaluation such as weakly relevant or strongly relevant: A possible numerical input value would be very questionable and scarcely reliable. Fuzzy logic seems to have the ideal solution to this problem. As is well known in the literature, fuzzy logic is ideal for formalizing incomplete and vague information [17].

The most natural manner to express qualitative information is to represent properties as linguistic variables. To make the interaction between a software engineer and the method as friendly as possible it is crucial to investigate how many and which primary terms would be meaningful for these linguistic variables. To this aim, we adopt the following method: we select a pool of software engineers and ask them to define the linguistic variables. Then, we stimulate a revision process within the pool aimed at reaching an agreement. Concerning Relevance and Autonomy of an entity, for instance, the pool concluded that property Relevance can be expressed as *weakly*, *slightly*, *fairly*, *substantially* and *strongly relevant*, and property Autonomy as *dependent*, *partially dependent* and *autonomous*. Figs. 2 and 3 show the meaning associated with the primary terms of Relevance and Autonomy. Here, standard piecewise quadratic functions are used to define membership functions. Consequently, artifact type Entity can be defined as:

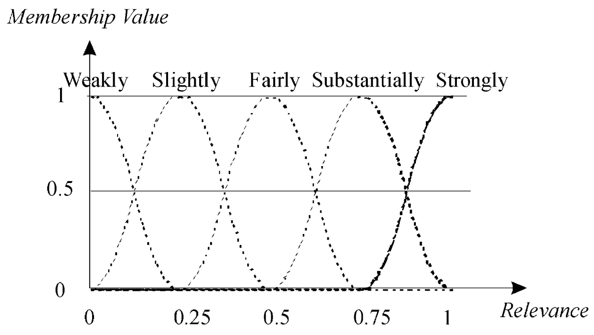


Fig. 2. Linguistic variable Relevance.

$\mathit{Name} \leftarrow [\textbf{\textit{Entity}}, (\textbf{\textit{Membership}}, [0, 1]), (\textbf{\textit{Relevance}},$

$\{\mathit{Weakly}, \mathit{Slightly}, \mathit{Fairly}, \mathit{Substantially}, \mathit{Strongly}\}),$

$(\textbf{\textit{Autonomy}}, \{\mathit{Dependent}, \mathit{Partially\ Dependent},$

$\mathit{Autonomous}\})]$

In general, we observed that software engineers tend to partition uniformly the universe of discourse and to use smooth membership functions to describe fuzzy sets.

4.4. Fuzzy methodological rules

Methodological rules have now to be reformulated as fuzzy rules. To express fuzzy rules in a convenient form, Membership is also transformed into a linguistic variable. The primary terms of Membership and the meanings associated with them are the same as those of property Relevance.

Consider the modified version of rule *Candidate Class Identification*:

IF AN ENTITY IN A REQUIREMENT SPECIFICATION IS *RELEVANCE VALUE* RELEVANT **AND** CAN EXIST *AUTONOMY VALUE* AUTONOMOUS IN THE APPLICATION

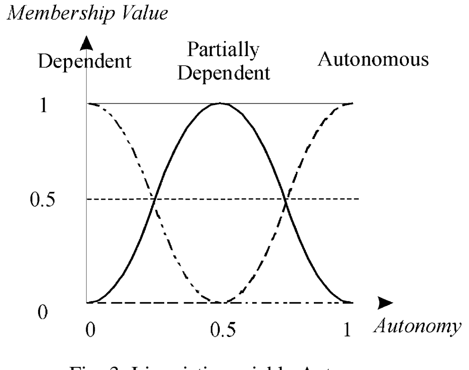


Fig. 3. Linguistic variable Autonomy.

Table 2
Sub-rules of rule *Candidate Class Identification*

P ← [Candidate Class: (Membership	*P* ← [Entity, (Relevance:				
	Weakly	Slightly	Fairly	Substantially	Strongly
Dependent	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Slightly*
Partially dependent	*Weakly*	*Slightly*	*Slightly*	*Fairly*	*Fairly*
Autonomous	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Strongly*
P ← [Entity, (Autonomy:					

DOMAIN **THEN** IT IS *MEMBERSHIP VALUE* A CANDIDATE CLASS.

Here, an entity and a candidate class are the artifact types to be reasoned, Relevance and Autonomy are the properties, and *relevance value* and *autonomy value* indicate the domains of these properties. Each combination of relevance and autonomy values of an entity has to be mapped into one of the five membership values to artifact type Candidate Class. The resulting 15 *sub-rules* are shown in Table 2. Each element of the table, shown in italics, represents the consequent part of the sub-rule. For example, if the relevance and autonomy values are respectively *strongly* and *autonomous*, then membership value to Candidate Class is *strongly*. Adopting the same notation as in Section 4.1, this sub-rule can also be represented as:

$$P \leftarrow [\textbf{\textit{Entity}}, (\textbf{\textit{Membership}} : 1)\ (\textbf{\textit{Relevance}} : \textit{Strongly}), (\textbf{\textit{Autonomy}} : \textit{Strongly})] \Rightarrow^{f} P \leftarrow [\textbf{\textit{CandidateClass}}, (\textit{Membership} : \textit{Strongly})]$$

Here, *P* indicates a variable, which is instantiated to the artifact being reasoned on, and $\Rightarrow^{f}$ represents a fuzzy implication operator. The value of Membership to Entity is 1 because Entity is considered as the starting artifact type.

The choice of the values shown in Table 2 is based on our intuition and knowledge of object-oriented methods [1–4]. Intuitively, the more an entity is autonomous and relevant, the more the entity is a candidate class. We can suppose that the membership value to Candidate Class can be computed as product of the relevance and autonomy values. The sub-rules shown in Table 2 have been generated based on this observation and on the meaning associated with each linguistic value. For instance, when an entity is weakly relevant and dependent on another entity, the entity has weakly the characteristics to be identified as a candidate class. With the increase of relevance and autonomy, the entity is more and more characterized as a candidate class.

In the following, we illustrate how the other classical methodological rules shown in Section 2.1 can be transformed into fuzzy rules. The fuzzy version of rule *Candidate Attribute Identification* is as follows:

IF AN ENTITY IN A REQUIREMENT SPECIFICATION IS *RELEVANCE VALUE* RELEVANT **AND** CAN EXIST *AUTONOMY VALUE* AUTONOMOUS IN THE APPLICATION DOMAIN, **THEN** IT IS *MEMBERSHIP VALUE* A CANDIDATE ATTRIBUTE.

The sub-rules corresponding to the rule are shown in Table 3 and are derived from the following intuition of artifact type Candidate Attribute: the more an entity is relevant and its existence is dependent on another entity, the more the entity is a candidate attribute.

As discussed in Section 4.2, if an appropriate combination of fuzzy implication, composition and aggregation operators, and suitable partitions of input and output spaces are chosen, the conclusion inferred from a set of fuzzy rules corresponds to the consequent part of the rule whose antecedent matches the fact. We would like to point out that the partitions shown in Figs. 2 and 3 are suitable partitions. If a software engineer inputs primary terms, the conclusion inferred from the rules is a primary term in its turn. For instance, if the software engineer decides that an entity is strongly relevant and autonomous, the entity is strongly a candidate class. This primary term may fire a chained rule (for instance, one of the conversion rules) and produce primary terms in its turn.

Table 3
Sub-rules of rule *Candidate Attribute Identification*

P ← [Candidate Attribute, (Membership:	*P* ← [Entity, (Relevance:				
	Weakly	Slightly	Fairly	Substantially	Strongly
Dependent	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Strongly*
Partially dependent	*Weakly*	*Slightly*	*Slightly*	*Fairly*	*Fairly*
Autonomous	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Slightly*
P ← [Entity, (Autonomy:					

Table 4
Sub-rules of rule *Attribute to Class Conversion*

	$P \leftarrow$ [Candidate Class, (Cohesion:, $P \leftarrow$ [Candidate Attribute, (Cohesion:				
$P \leftarrow$ [Class, (Membership:	Weakly	Slightly	Fairly	Substantially	Strongly
{Weakly, Weakly}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Substantially*
{Weakly, Slightly}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Substantially*
{Weakly, Fairly}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Strongly*
{Weakly, Substantially}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Strongly*
{Slightly, Weakly}	*Weakly*	*Weakly*	*Slightly*	*Fairly*	*Substantially*
{Slightly, Slightly}	*Weakly*	*Slightly*	*Fairly*	*Fairly*	*Substantially*
{Slightly, Strongly}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Strongly*
{Fairly, Weakly}	*Weakly*	*Weakly*	*Slightly*	*Fairly*	*Substantially*
{Fairly, Fairly}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Strongly*
{Substantially, Weakly}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Substantially*
{Strongly, Slightly}	*Weakly*	*Slightly*	*Fairly*	*Fairly*	*Substantially*
{$P \leftarrow$ [Candidate Attribute, (Membership:, $P \leftarrow$ [Candidate Class, (Membership:}					

To transform the two conversion rules, we need to analyze their meaning in detail. Let us consider rule *Attribute to Class Conversion*. This rule reasons on the property of an artifact of being responsible for a set of functions, but captures only the following part of the intuition derivable from this property: The more an attribute is responsible for a set of functions, the more it is a class. There exists, however, another part of intuition concerning the property: the more an artifact is a candidate class and the more is responsible for functions, the more it is a class. As in current methods, an artifact is either a candidate class or a candidate attribute or neither, this part of intuition is not relevant. Indeed, if the artifact is already a candidate class, to be responsible for a set of functions can only confirm that the artifact is a class. If the artifact is a candidate attribute, a part of intuition is implemented by rule *Attribute to Class Conversion*. If it is neither a candidate attribute nor a candidate class, the artifact is practically discarded and therefore its responsibility for functions is not evaluated. On the contrary, in the fuzzy method, an artifact can be a partial instance of both candidate attribute and candidate class at the same time. It follows that the overall intuition can be captured and modeled. Thus, the fuzzy version of rule *Attribute to Class Conversion* has the membership values to both candidate class and candidate attribute as inputs. The fuzzy version of rule *Attribute to Class Conversion* is as follows:

IF *P* IS *MEMBERSHIP VALUE* A CANDIDATE ATTRIBUTE **AND** *P* IS *MEMBERSHIP VALUE* A CANDIDATE CLASS **AND** OPERATIONS BELONG TO P *COHESION VALUE* **THEN** P IS *MEMBERSHIP VALUE* A CLASS

Property Cohesion has the same primary terms as Membership. The antecedent of this rule has three input linguistic variables. To represent the sub-rules of this rule, we still adopt a tabular representation, but each row of the table corresponds to one of the possible combinations of the primary terms of two input linguistic variables, and each column to one of the primary terms of the remaining linguistic variable. To reduce the number of rows and simplify the representation, we suppose that a software engineer can input only primary terms for rules *Candidate Attribute Identification* and *Candidate Class Identification*. This implies that the conclusions inferred from these rules are primary terms of Membership to candidate attribute and candidate class. From the definitions given in Tables 2 and 3, it can be deduced that only 11 combinations of membership values to candidate class and candidate attribute are possible. Table 4 defines the sub-rules of rule *Attribute to Class Conversion* under this assumption. Here, the rows indicate the pairs of possible membership values to candidate attribute and candidate class, and the columns the values of property Cohesion. Note that Cohesion is a property of both Candidate Class and Candidate Attribute. Actually, property Cohesion is one of those properties that determine the inconsistency between Class and Attribute. The definition of the sub-rules takes the following intuitive aspects into account.

1. The more a set of functions belongs to *P*, the more *P* is a class independently of the membership value of *P* to candidate attribute.
2. The more *P* is a candidate class, the more *P* is a class.

The first aspect enables the functional view to possibly reverse the judgment expressed by the relevance and autonomy view analyzed by the first two rules. Although an entity can be considered to be weakly relevant in the application domain and judged to be both weakly a candidate attribute and weakly a candidate class, the functional view can reverse that initial judgment and transform the entity into

Table 5
Sub-rules of rule *Class to Attribute Conversion*

	$P \leftarrow$ [Candidate Class, (Cohesion:, $P \leftarrow$ [Candidate Attribute, (Cohesion:				
$P \leftarrow$ [Attribute, (Membership:	Weakly	Slightly	Fairly	Substantially	Strongly
{Weakly, Weakly}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Weakly, Slightly}	*Slightly*	*Slightly*	*Weakly*	*Weakly*	*Weakly*
{Weakly, Fairly}	*Fairly*	*Slightly*	*Weakly*	*Weakly*	*Weakly*
{Weakly, Substantially}	*Substantially*	*Fairly*	*Slightly*	*Weakly*	*Weakly*
{Slightly, Weakly}	*Slightly*	*Slightly*	*Slightly*	*Weakly*	*Weakly*
{Slightly, Slightly}	*Slightly*	*Slightly*	*Slightly*	*Weakly*	*Weakly*
{Slightly, Strongly}	*Substantially*	*Substantially*	*Fairly*	*Weakly*	*Weakly*
{Fairly, Weakly}	*Fairly*	*Fairly*	*Slightly*	*Slightly*	*Weakly*
{Fairly, Fairly}	*Substantially*	*Fairly*	*Fairly*	*Slightly*	*Weakly*
{Substantially, Weakly}	*Substantially*	*Substantially*	*Fairly*	*Slightly*	*Weakly*
{Strongly, Slightly}	*Strongly*	*Substantially*	*Fairly*	*Slightly*	*Weakly*
{$P \leftarrow$ [Candidate Attribute, (Membership:, $P \leftarrow$ [Candidate Class, (Membership:}					

a substantial class. We observe that non-relevant entities are discarded in current methods without any further investigation.

Let us consider now rule *Class to Attribute Conversion*. This rule reasons on the property of an artifact of being responsible for a set of functions, but captures only the following part of the intuition derivable from this property: The less a class is responsible for a set of functions, the more it is an attribute. There exists, however, another part of intuition concerning the property: the more an artifact is a candidate attribute and the less is responsible for functions, the more it is an attribute. As explained above, the overall intuition cannot be implemented in current methods due to consistency constraints. This intuition can successfully be modeled in fuzzy logic. The fuzzy version of rule *Class to Attribute Conversion* is as follows:

IF *P* IS *MEMBERSHIP VALUE* A CANDIDATE ATTRIBUTE **AND** *P* IS *MEMBERSHIP VALUE* A CANDIDATE CLASS **AND** OPERATIONS BELONG TO *P COHESION VALUE* **THEN** *P* IS *MEMBERSHIP VALUE* AN ATTRIBUTE

Table 5 defines the sub-rules of rule *Class to Attribute Conversion*. The definition of the sub-rules takes the following aspects into account.

1. The less a set of functions belongs to *P* and the more *P* is a candidate class, the more *P* is an attribute.
2. The less a set of functions belongs to *P* and the more *P* is a candidate attribute, the more *P* is an attribute.
3. The less *P* is a candidate class and a candidate attribute, the less *P* is an attribute independently of how much a set of functions belongs to *P*.

After identifying classes, rules *Aggregation Identification* and *Inheritance Identification* determine whether and which relation exists between classes. Obviously, the value of membership of a relation to the set of aggregation or inheritance relations depends on the values of membership of the artifacts being reasoned to class. The fuzzy version of rule *Aggregation Identification* is as follows:

IF P_1 IS *MEMBERSHIP VALUE* A CLASS **AND** P_2 IS *MEMBERSHIP VALUE* A CLASS **AND** P_1 *CONTAINMENT VALUE* CONTAINS P_2 **THEN** RELATION BETWEEN P_1 AND P_2 IS *MEMBERSHIP VALUE* AN AGGREGATION.

Property Containment has the same primary terms as Membership. As the antecedent of this rule has three input linguistic variables, we adopt the tabular representation with each row that corresponds to one of the possible combinations of primary terms of two input linguistic variables. Table 6 shows the sub-rules of rule *Aggregation Identification*. Here, the rows indicate the pairs of possible membership values of P_1 and P_2 to Class, and the columns the values of property Containment. Sub-rules are defined based on the following intuition: the more P_1 and P_2 are classes and P_1 contains P_2, the more the relation between P_1 and P_2 is an aggregation.

Similar to rule *Aggregation Identification*, the fuzzy version of rule *Inheritance Identification* is as follows:

IF P_1 IS *MEMBERSHIP VALUE* A CLASS **AND** P_2 IS *MEMBERSHIP VALUE* A CLASS **AND** P_2 *IS-A-KIND-OF VALUE* IS-A-KIND-OF P_1 **THEN** RELATION BETWEEN P_1 AND P_2 IS *MEMBERSHIP VALUE* AN INHERITANCE.

Property Is-a-kind-of has the same primary terms as Membership. The definition of the sub-rules of rule *Inheritance Identification* can be obtained replacing the property Containment with the property Is-a-kind-of in Table 6.

Rule *Inheritance Modification* is applied to verify the complexity of a hierarchy. Here, we suppose that the hierarchy has been already defined and the number of

Table 6
Sub-rules of rule *Aggregation Identification*

	$P_1 \leftarrow$ [Class, (Containment:				
$P_3 \leftarrow$ [Inheritance, (Membership:	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Strongly*
{Weakly, Weakly}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Weakly, Slightly}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Weakly, Fairly}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Weakly, Substantially}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Weakly, Strongly}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Slightly, Weakly}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Slightly, Slightly}	*Weakly*	*Slightly*	*Slightly*	*Slightly*	*Slightly*
{Slightly, Fairly}	*Weakly*	*Slightly*	*Slightly*	*Slightly*	*Slightly*
{Slightly, Substantially}	*Weakly*	*Slightly*	*Slightly*	*Slightly*	*Slightly*
{Slightly, Strongly}	*Weakly*	*Slightly*	*Slightly*	*Slightly*	*Slightly*
{Fairly, Weakly}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Fairly, Slightly}	*Weakly*	*Slightly*	*Slightly*	*Slightly*	*Slightly*
{Fairly, Fairly}	*Weakly*	*Slightly*	*Fairly*	*Fairly*	*Fairly*
{Fairly, Substantially}	*Weakly*	*Slightly*	*Fairly*	*Fairly*	*Fairly*
{Fairly, Strongly}	*Weakly*	*Slightly*	*Fairly*	*Fairly*	*Fairly*
{Substantially, Weakly}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Substantially, Slightly}	*Weakly*	*Slightly*	*Slightly*	*Slightly*	*Slightly*
{Substantially, Fairly}	*Weakly*	*Slightly*	*Fairly*	*Fairly*	*Fairly*
{Substantially, Substantially}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Substantially*
{Substantially, Strongly}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Substantially*
{Strongly, Weakly}	*Weakly*	*Weakly*	*Weakly*	*Weakly*	*Weakly*
{Strongly, Slightly}	*Weakly*	*Slightly*	*Slightly*	*Slightly*	*Slightly*
{Strongly, Fairly}	*Weakly*	*Slightly*	*Fairly*	*Fairly*	*Fairly*
{Strongly, Substantially}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Substantially*
{Strongly, Strongly}	*Weakly*	*Slightly*	*Fairly*	*Substantially*	*Strongly*
{$P_1 \leftarrow$ [Class, (Membership:, $P_2 \leftarrow$ [Class, (Membership:					

immediate subclasses can be quantified. The fuzzy version of rule *Inheritance Modification* is as follows:

IN THE CLASS HIERARCHY, **IF** THE NUMBER OF IMMEDIATE SUBCLASSES SUBORDINATED TO A CLASS IS *SUBCLASSES NUMBER*, **THEN** THE INHERITANCE HIERARCHY IS *COMPLEXITY VALUE*.

The primary terms of linguistic variables Subclasses Number and Complexity are *low*, *medium* and *high*. Fig. 4 shows the definition of Subclasses Number and Complexity. Table 7 shows the three sub-rules derived from this rule. While in the first phases of the development process, the software engineer can input only qualitative information; in the later phases, this information can be more precise. The software engineer can, for instance, input crisp values to rule *Inheritance Modification*. In this case, the generalized modus ponens discussed in Section 4.2 can be still applied by considering the crisp value as a fuzzy set which is characterized by a membership value equal to 1 in correspondence to the crisp value itself and 0 otherwise. It follows that fuzzy logic can manage both crisp and linguistic values within the same framework.

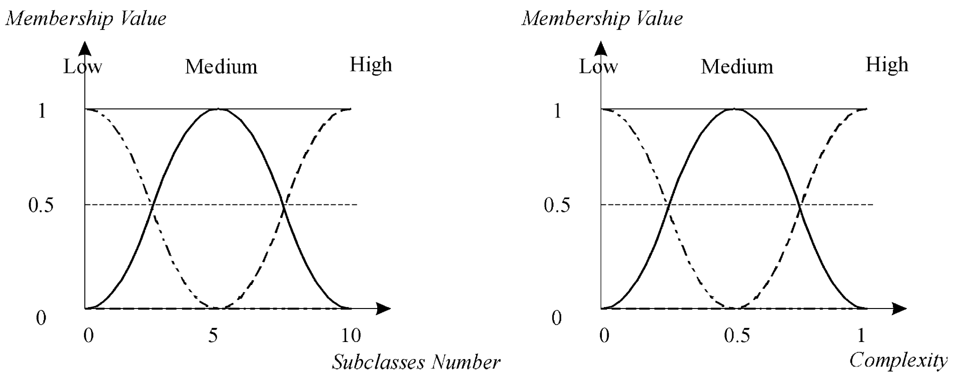


Fig. 4. Linguistic variables Subclasses Number and Complexity.

Table 7
Sub-rules of rule *Inheritance Modification*

	P ← [Inheritance, (Subclasses Number:		
	Low	*Medium*	*High*
P ← [Inheritance, (Complexity:	Low	Medium	High

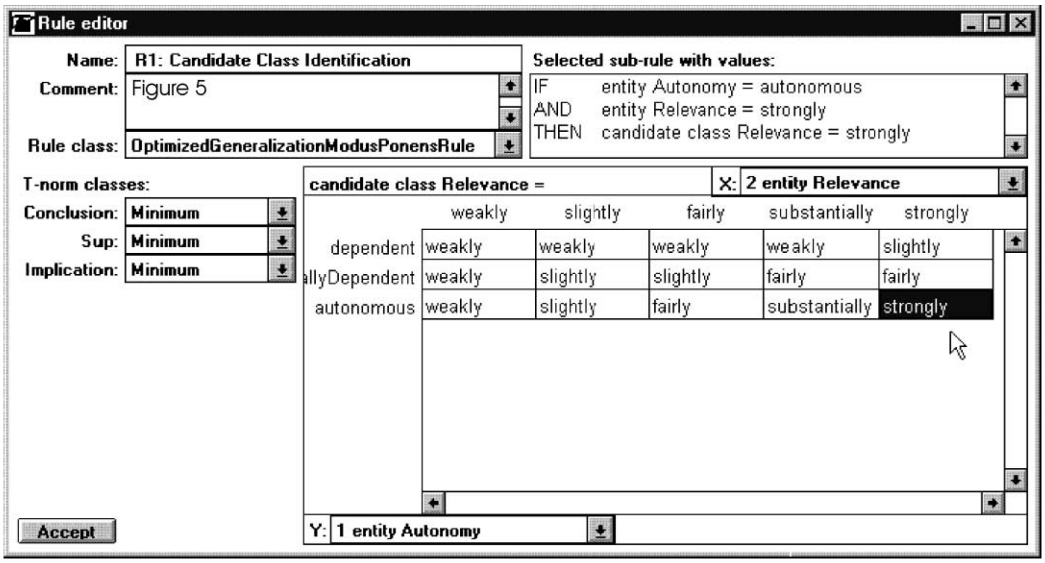
In conclusion, the use of fuzzy methodological rules seems to be very natural and software engineers should not have a lot of trouble to migrate to this new approach. This conviction is supported by the rapid success that fuzzy logic has reached in other fields. For instance, in control applications, where fuzzy logic has been widely and successfully used in the last years, control engineers consider very natural to codify their experience using fuzzy rules [20]. As methodological rules are quite close to control rules, we expect that software engineers feel the same naturalness as control engineers in using fuzzy logic. Further, software engineers do not need to know specific details of fuzzy inference, but they need only to be conscious that approximate reasoning is able to reproduce quite faithfully their reasoning. Finally, software engineers should be particularly stimulated by the possibility to express their perception of an artifact using their natural language without being repressed by the necessity to fit restrictive input values imposed by methodological rules.

If fuzzy logic-based methodological rules are applied, no alternative design solutions are theoretically eliminated. Software engineers are not forced to take abrupt decisions, but are encouraged to express their perception of artifacts in their natural language. This reduces the loss of information and increases the quality of the whole development process. Further, perceptions expressed as linguistic expressions can be used as measures of alternatives and exploited to resolve inconsistencies whenever it is necessary. The complexity of this concurrent analysis of multiple alternative solutions is managed by using an appropriately designed CASE environment.

5. CASE environment

Our CASE environment is based on Rational Rose™ [25] because of its availability in our laboratory. We developed a separate repository to store the extended artifacts. We linked these artifacts to the Rational Rose™ environment using the OLE™ technology. A version of this tool is presented in Ref. [28].

We built tools to support method engineers in codifying properties of artifact types as linguistic variables and in defining fuzzy rules. To define a linguistic variable, the method engineer is required to input the universe of discourse, the set of the primary terms and the membership functions associated with each primary term. At the present, our tool allows a method engineer to define rectangular, trapezoidal, triangular and standard piecewise quadratic membership functions. The linguistic variables are used to define the fuzzy rules. The rule editor is shown in Fig. 5. Here, the definition of the fuzzy rule *Candidate Class Identification* is illustrated. A rule is defined by providing its name, the table that describes the sub-rules, and the parameters to select an appropriate implementation of fuzzy reasoning. The menu Rule Class allows selecting the type of implication operator. The most used types of implication operators have been implemented in the tool: for each type, four different implication operators can be derived by selecting one of the four different triangular norms (T-norms) shown in Table 1 in the 'Implication' menu of section 'T-norm classes'. In this section, the menus 'Sup' and 'Conclusion' allow a method engineer to select the



Name: R1: Candidate Class Identification
Comment: Figure 5
Rule class: OptimizedGeneralizationModusPonensRule

Selected sub-rule with values:
IF entity Autonomy = autonomous
AND entity Relevance = strongly
THEN candidate class Relevance = strongly

T-norm classes:
Conclusion: Minimum
Sup: Minimum
Implication: Minimum

candidate class Relevance = X: 2 entity Relevance

	weakly	slightly	fairly	substantially	strongly
dependent	weakly	weakly	weakly	weakly	slightly
llyDependent	weakly	slightly	slightly	fairly	fairly
autonomous	weakly	slightly	fairly	substantially	strongly

Y: 1 entity Autonomy

Fig. 5. Tool for defining the fuzzy rules.

T-norm adopted in the composition and aggregation operators, respectively. For each combination of the primary terms of the properties involved in the antecedent of the rule, the method engineer can choose one of the primary terms of the property involved in the consequent part. The properties used in the rules have to be preliminarily defined as linguistic variables. Although Fig. 5 shows the definition of rules with antecedents composed only by two propositions, the tool allows defining antecedents with any number of propositions. In this case, the method engineer can define the sub-rules for each possible combination of the primary terms of the input linguistic variables in the following way: from time to time, if N is the number of input linguistic variables, he/she fixes the values of $N-2$ linguistic variables and defines all sub-rules generated by the possible combinations of the primary terms of the remaining two linguistic variables using the rule editor in Fig. 5.

During the execution of the rules, the software engineer interacts with the tool, which is shown in Fig. 6. Here, the software engineer is requested to provide the Relevance value used in fuzzy rule *Candidate Class Identification*. The software engineer can select one of the primary terms, input a linguistic value among those allowed by the grammar described in Section 4.2, or input a numerical value. To 'tune' the software engineer's interpretation of each primary term to the method engineer's interpretation, the tool shows the membership functions, which define the primary terms. Each new input provided by the software engineer infers a number of sub-rules: for instance, the value of Relevance of an entity in the requirement specification infers both the sub-rules defined for instantiating the entity as a candidate class and those for instantiating the entity as a candidate attribute. In this way, each path of the development process is investigated concurrently and different design solutions can be analyzed at the same time.

To reduce the complexity of the development process, the tool allows a software engineer to fix a threshold on the

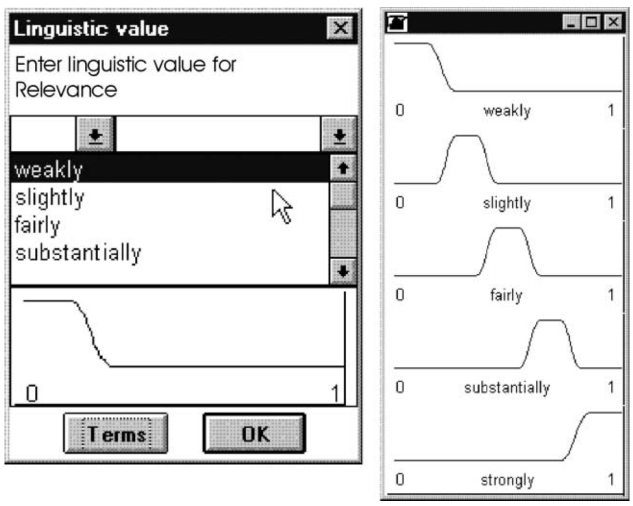


Fig. 6. Tool for providing the linguistic values.

membership values: if an artifact is an instance of an artifact type with membership value below the threshold, then the reasoning paths involving that artifact type are not investigated for that artifact. The value of the threshold can be numeric or linguistic. For example, a software engineer could decide that instances belonging less than slightly to an artifact type are discarded in the development process. Thus, if an entity is selected as weakly an attribute, no rule, which reasons on artifact type Attribute, will be inferred for that entity. In addition, the tool allows a software engineer to fix priorities in investigating conflicting alternatives. For instance, the software engineer can establish to be guided to work first on the alternative with the highest measure and then on the others. These options allow software engineers to reach a compromise between complexity and accuracy.

6. Inconsistency resolution

During the development process, each software development element (SDE) involved in the inference process maintains its story, that is, the values of membership to each artifact type reasoned so far. Each SDE knows which artifact types are in conflict: for instance, an SDE knows that it cannot be a class and an attribute at the same time. Conflicting artifact types are identified by using the definition given in Section 4.1. Inconsistencies are not resolved until explicitly requested. This occurs, for instance, when the product of a development phase has to be released. When such a request is made, each SDE applies an inconsistency resolution policy. Policies depend on the inconsistency type and may be affected by contextual factors such as application type, sensibility and experience of the software engineer and desired level of quality. Typically, an inconsistency resolution policy is implemented as comparing defuzzified membership values and selecting the artifact corresponding to the highest value. If the highest value is above an 'existence' threshold (typically fixed to 0.5), the artifact is included into final product; otherwise it is eliminated. The existence threshold allows eliminating for example classes or attributes which derive from weakly or slightly relevant entities, and are not revalued by the application of the conversion rules shown in Section 4.4. Optionally, before the selection or the elimination is made the software engineer may be consulted.

In our CASE tool, this consulting is activated if the compared defuzzified values are close to each other or if the value of the winning artifact type is close to the existence threshold. Software engineers may take a decision based on their experience and perception of the application domain. Let us suppose, for instance, that the fuzzy logic-based method has selected an entity both as a class and as an attribute with similar membership degree. If conceptual modeling is considered important, then the software engineer may decide to implement the entity as an attribute. If reusability is the main concern, then class may be the choice. In this case, the goal of the application and the experience of the software engineer affect the resolution policy.

Resolution of an inconsistency may trigger a 'chain reaction' of revisions of membership values. Let us suppose that an entity A has been considered as slightly an attribute and substantially a class. Let us assume that the adopted resolution policy is to select an artifact as instance of the artifact type corresponding to the highest membership value. Entity A is, therefore, selected as a class. This implies that the membership value of A to artifact type Class is now 1. All membership values that depend on this value have to be revalued. For instance, all values of membership to Inheritance or Aggregation relations, which involve A, will be recomputed. The new membership values may affect the result of the inconsistency resolution policy applied to Inheritance and Aggregation relations. It is obvious that the order of application of inconsistency resolution policies is critical to obtain meaningful products. This order has to follow the order defined by the methodological rules and used to identify artifacts during the development process.

7. Evaluation with respect to the requirements

In this section we evaluate our approach with respect to the requirements presented in Section 3.

- *Reducing the quantization error*: Increasing the number of possible values for properties of artifact types and consequently the number of quantization levels, fuzzy logic methodological rules reduce the quantization error [22]. Further, in the fuzzy logic-based method, the accumulation of the quantization error during the software development process is much less than the accumulation of error in the classical logic-based method [23]. The improvement is achieved by capturing as much as possible the software engineer's perception.
  In the current methods, inconsistencies are resolved during the application of each rule. It follows that the resulting object model is less adaptable to changes and will not adapt itself to the new information available during the software development process. In the fuzzy logic-based method, inconsistencies are left and therefore, in principle, none of the artifacts is eliminated. The fuzzy logic-based method can be considered as a learning process; a new aspect of the problem being considered is learned after the application of each rule. Obviously, a new aspect can modify the previously gathered property values. The fuzzy logic theory provides techniques to reason and compose the results of the rules. Clearly, software development through learning creates very adaptable and reusable design models.
- *Provide a measure for the alternatives*: The fuzzy logic-based method allows inconsistencies and associates a measure for each inconsistent alternative. Measures can be expressed as linguistic or crisp values and are useful when the inconsistencies have to be resolved. Artifact measures are adapted through various phases. In the end, these measures are used to resolve inconsistencies.
- *Manage complexity*: In our approach, leaving inconsistency allows to investigate design alternatives concurrently. This improves the quality of the software development process but also increases its complexity. The software engineer, however, can establish a threshold on membership values of an artifact to an artifact type: artifacts which have a membership value to an artifact type below the threshold do not trigger the rules which reason on that artifact type. This reduces the design space. As described in Section 5, various automatic control mechanisms can be established to manage the complexity. For example, in case of inconsistent alternatives, the software engineer may be guided to work on the alternative with the highest measure. If this measure decreases in the subsequent phases, other alternatives can be automatically brought to the attention of the software engineer. Our experience with the experimental CASE environment shows that being able to process rules and inconsistencies automatically hides the internal complexity. Concluding, the fuzzy logic-based approach provides a unique opportunity to tune the quality of CASE environments with respect to the memory and processing costs.

8. Related work

In the following, we briefly present the related work on inconsistency management and application of fuzzy logic to software engineering.

8.1. Inconsistency management

The need of tolerating inconsistencies during software development has been pioneered in Ref. [5]. Here, inconsistent data are automatically marked by means of *pollution markers*. A pollution marker makes the inconsistent data known to procedures or human agents, which are responsible for solving the inconsistency. Further, it protects the inconsistent data from the action of other procedures sensitive to the inconsistency.

In Ref. [14], inconsistency handling in multi-perspective specifications is studied by using the ViewPoint framework [24]. In this framework, each developer specifies the system by using a representation language and a development process according to his/her own viewpoint. The consistency rules are expressed in terms of classical logic and represent some of the implicit assumptions and integrity constraints used in controlling and coordinating a set of viewpoints. A meta-language based on linear-time temporal logic is used to specify the actions necessary to cope with inconsistency. In Ref. [15,16], the approach of inconsistency handling shown in Ref. [14] is further developed by introducing quasi-classical logic. Unlike classical logic, quasi-classical logic permits the derivation of non-trivial inferences from inconsistent information. In the presence of inconsistencies, this allows limited reasoning and consequently the possibility of analyzing such inconsistencies. The analysis may identify the sources of inconsistency and qualify the inconsistent information.

By analogy with the viewpoints for multi-perspective specifications discussed above, in Ref. [27] the authors propose process viewpoints to manage process inconsistency. Process inconsistencies are interpreted as differences of various kinds in the ways in that different people perceive or execute a process. Viewpoints provide multi-perspective descriptions of software processes and allow highlighting different perceptions of a software process. Process inconsistencies can therefore be detected and used to stimulate a resolution process, which can either remove the inconsistencies, or flag them and ensure that the reason for their existence is understood by the process participants.

In Ref. [11], inconsistencies, which occur between definition and actual instance of a development process, have been studied in human-centered systems. It is argued that processes defining the interaction between humans and computerized tools have to tolerate, control and support inconsistencies and deviations of real-world behaviors with respect to the process model. This is necessary to maintain an effective flexibility and adaptability to the evolving needs and preferences of the humans. They propose a framework for formally defining the concepts of inconsistency and deviation between a human-centered-system and its process support system. Deviations are tolerated as long as they do not affect the correctness of the system. Then, a *reconciling sequence* of feasible events starting from an inconsistent state and ending in a consistent state has to be executed. An evolution of this work coping with the ability of tolerating deviations from the process model during enactment and supporting users in reconciling the process model with actual process is described in Ref. [12].

Our work is similar to the related work presented in this section in that tolerating and coping with inconsistency are considered important in creating flexible software systems. Further, several analogies exist with respect to the classical logic-based techniques used to detect inconsistencies. Differences may be perceived on the semantics of the inconsistencies taken into account, on how the inconsistencies are handled during the development process and on how they are resolved when a product has to be released.

8.2. Applications of fuzzy logic to software engineering

To the best of our knowledge, a few papers have investigated the use of fuzzy logic in software engineering. In Ref. [13], fuzzy techniques are used to handle the uncertainty arisen from the classification of components and their retrieval for reuse according to software behavioral properties.

Benedicenti et al. exploit fuzzy logic for coping with unreliable data in a business process modeling method [6]. Fuzzy logic is employed to attribute resources to activities, determine activity cost driver and resource (per activity) cost driver.

Liu and Yen propose a systematic approach for specifying and analyzing imprecise requirements [21]. The constraint imposed by an imprecise requirement R is represented as a satisfaction function that maps an element of R's domain D to a number in [0,1]. In practice, the satisfaction function defines a fuzzy subset of D that satisfies the imprecise requirement. Requirements are expressed in the canonical form of Zadeh's test core semantics. Based on different impact of satisfying a requirement on the satisfaction degree of another requirement, four types of relationships between requirements are introduced: conflicting, cooperative, mutually exclusive and irrelevant. For instance, two imprecise requirements are said to be conflicting with each other if an increase in the satisfaction degree of one requirement decreases the satisfaction degree of the other. These formal definitions permit to develop knowledge-based techniques, which allow assessing the impact of requirement changes and inferring relationships between requirements in order to detect implicit conflicts. Detected conflicts are therefore resolved based on the priority associated with each requirement: satisfaction of requirements with high priority is preferred to the satisfaction of requirements with low priority. The priority is assigned based on the marginal rate of substitution, i.e. the maximal amount of a decision attribute a customer decides to sacrifice for a unit increase in another decision attribute.

Cîmpman and Oquendo propose to use fuzzy logic to monitor software processes [8–10]. The monitoring process focuses on the detection of deviations between the actual enacting process and the process enactment plan. The level of deviation is computed for different aspects of the process like progress, cost, structure (order between activities), etc. and varies from total conformance to no conformance at all. The monitoring system is a part of a qualitative control system, which assists the process manager during the process evolution: the control system detects deviations and indicates possible corrective actions. Fuzzy logic is used to represent possible imprecise and uncertain information handled by the monitoring system, and to reason on it. Similarly to our approach, the use of fuzzy logic is justified by its ability to naturally represent uncertain and imprecise information, and to constitute a good framework for approximate reasoning. Whereas our approach, however, focuses on managing inconsistencies of software products, the monitoring system focuses on handling deviations of software processes: the two approaches therefore act at two different abstraction levels.

9. Conclusions

Due to growing complexity of the today's software systems, identifying and managing inconsistencies are becoming crucial issues in software development. Current software development methods consider inconsistencies undesirable and try to resolve them whenever they are detected. To enforce consistency at all times in the development process can, however, result in loss of information and excessive restriction of the design space. Based on this observation, new approaches have been proposed in the last years. They consider inconsistencies as useful *components* of the development process and therefore adopt techniques to tolerate them as long as needed.

Within such a context, we have proposed a fuzzy logic based approach to model inconsistencies during the software development process. We have shown that fuzzy logic-based techniques could model inconsistencies effectively without altering the intuitive expressiveness of the current methods. Unlike other techniques used successfully to cope with inconsistencies, fuzzy logic offers a unique opportunity to model methodological rules and handle inconsistencies within the same framework. Linguistic variables allow capturing as much as possible the software engineer's perception in a natural way. Approximate reasoning permits to reason on the linguistic expressions to deduce conclusions and conduct the development process. Each rule determines to which extent an artifact is an instance of an artifact type and this membership degree can be considered as a measure of each alternative. Such a measure is useful in providing the software engineer with a means to control the complexity of the development process, and in defining policies for resolving inconsistencies when needed.

A small fuzzy logic-based method has been implemented using our experimental CASE environment and tested on an example problem [2]. We have observed that effectively the resulting object model is more adaptable than the one developed by using standard methods. We are currently developing a fuzzy logic-based method derived from the heuristics of the most popular object-oriented methods such as OMT [26]. The transformation of the heuristics into fuzzy methodological rules is based on our experience in developing object-oriented systems [1–4] and on interviews with experienced software engineers.

Acknowledgements

We thank Dr Klaas van den Berg for his comments on an earlier version of this paper.

References

[1] M. Aksit, L. Bergmans, Obstacles in object-oriented software development, Proc. OOPSLA '92, ACM SIGPLAN Notices 27 (10) (1992) 341–358.
[2] M. Aksit, F. Marcelloni, Deferring elimination of design alternatives in object-oriented methods, Concurrency and Computation — Practice and Experience, Wiley, New York, 2001 to be published.
[3] M. Aksit, F. Marcelloni, B. Tekinerdogan, Developing O.O. Frameworks using domain models, ACM Comput. Surv. 32 (1es) (2000).
[4] M. Aksit, B. Tekinerdogan, F. Marcelloni, L. Bergmans, Deriving object-oriented frameworks from domain knowledge, in: M. Fayad, D. Schmidt (Eds.), Object-Oriented Frameworks, Wiley, New York, 1999, pp. 169–198.
[5] R. Balzer, Tolerating Inconsistency, Proceedings of 13th International Conference on Software Engineering, Austin, Texas 1991 pp. 158–163.
[6] L. Benedicenti, G. Succi, T. Vernazza, A. Valerio, Object Oriented Process Modeling with Fuzzy Logic, Proceedings of the 1998 ACM symposium on Applied Computing 1998 pp. 267–271.
[7] S.R. Chidamber, C.F. Kemerer, A metrics suite for object-oriented design, IEEE Trans. Soft. Engng 20 (6) (1994) 476–492.
[8] S. Cîmpan, F. Oquendo, Dealing with software process deviations using fuzzy logic based monitoring, ACM Appl. Comput. Rev. 8 (2) (2000) 3–13.
[9] S. Cîmpan, F. Oquendo, Fuzzy Indicators for Monitoring Software Processes, Proceedings of the 6th European Workshop on Software Process Technology, Springer, Berlin, Germany, 1998 pp. 43–59.
[10] S. Cîmpan, F. Oquendo, On the Application of Fuzzy Set Theory to the Monitoring of Software-Intensive Processes, Proceedings of the Eighth International Fuzzy Systems Association World Congress, Nat. Central Univ, Chungli, Taiwan, 1999 pp. 1071–1076.
[11] G. Cugola, E. Di Nitto, A. Fuggetta, C. Ghezzi, Framework for formalizing inconsistencies and deviations in human-centered systems, ACM Trans. Soft. Engng Methodol. 5 (3) (1996) 191–230.
[12] G. Cugola, Tolerating deviations in process support systems via flexible enactment of process models, IEEE Trans. Soft. Engng 24 (11) (1998) 982–1001.
[13] E. Damiani, M.G. Fugini, Fuzzy Techniques for Software Reuse, Proceedings of the 1996 ACM Symposium on Applied Computing, ACM Philadelphia, PA, 1996, February (17–19) pp. 552–557.
[14] A.C.W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, B. Nuseibeh, Inconsistency handling in multiperspective specifications, IEEE Trans. Soft. Engnr. 20 (8) (1994) 569–578.
[15] A. Hunter, B. Nuseibeh, Analysing Inconsistent Specifications, Proceedings of 3rd International Symposium on Requirements Engineering, IEEE CS Press, Annapolis, USA, 1997.
[16] A. Hunter, B. Nuseibeh, Managing inconsistent specifications: reasoning, analysis, and action, ACM Trans. Soft. Engng Methodol. 7 (4) (1998) 335–367.
[17] G.J. Klir, B. Yuan, Fuzzy Sets and Fuzzy Logic — Theory and Applications, Prentice-Hall, Upper Saddle River, NJ, 1995.
[18] G. Lakoff, Women, Fire, and Dangerous Things, The University of Chicago Press, Chicago, 1987.
[19] B. Lazzerini, F. Marcelloni, Some considerations on input and output partitions to produce meaningful conclusions in fuzzy inference, Fuzzy Sets Syst. 113 (2) (2000) 221–235.
[20] C.C. Lee, Fuzzy logic in control systems: fuzzy logic controller, Part II, IEEE Trans. Syst., Man, Cybernet. 20 (2) (1990) 419–435.
[21] X.F. Liu, J. Yen, An analytic framework for specifying and analyzing imprecise requirements, Proceedings of the 18th International Conference on Software Engineering 1996 pp. 60–69.
[22] F. Marcelloni, M. Aksit, Reducing Quantization Error and Contextual Bias Problems in Software Development Processes by Applying Fuzzy Logic, Proceedings of NAFIPS'99, IEEE Press, New York, 1999 pp. 268–272.

[23] F. Marcelloni, M. Aksit, Improving object-oriented methods by using fuzzy logic, ACM Appl. Comput. Rev. 8 (2) (2000) 14–23.
[24] B. Nuseibeh, J. Kramer, A.C.W. Finkelstein, A framework for expressing the relationships between multiple views in requirements specifications, IEEE Trans. Soft. Engnr 20 (10) (1994) 760–773.
[25] Rational Rose, url: http://www.rational.com/products/rose/.
[26] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Englewood cliffs, NJ, 1991.
[27] I. Sommerville, P. Sawyer, S. Viller, Managing process inconsistency using viewpoints, IEEE Trans. Soft. Engng 25 (6) (1999) 784–799.
[28] B. Tekinerdogan, M. Aksit, Modeling Heuristic Rules of Methods, Department of Computer Science, University of Twente, Twente, 1998.
[29] I.B. Turksen, Y. Tian, Combination of rules or their consequences in fuzzy expert systems, Fuzzy Sets Syst. 58 (1993) 3–40.
[30] A. Van Lamsweerde, R. Darimont, E. Letier, Managing conflicts in goal-driven requirements engineering, IEEE Trans. Soft. Engng 24 (11) (1998) 908–926.
[31] E. Yourdon, Modern Structured Analysis, Yourdon Press, Englewood Cliffs, NJ, 1989.
[32] L.A. Zadeh, Outline of a new approach to the analysis of complex systems and decision processes, IEEE Trans. Syst., Man, Cybernet. SMC-3 (1) (1973) 28–44.
[33] L.A. Zadeh, Fuzzy Logic = Computing with Words, IEEE Trans. Fuzzy Syst. 4 (2) (1996) 103–111.