FUZZYNESS AND IMPRECISION IN SOFTWARE ENGINEERING

M. Burgin
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA

N. Debnath
Department of Computer Science
Winona State University
Winona, MN 55987, USA

Abstract

In this paper, we study how fuzziness in software engineering emerges and how to reflect this fuzziness measuring software qualities. Principal means in these processes are software metrics with values in categorical data represented by software metrics with values in fuzzy sets and linguistic variables. This study is aimed to support the development of high quality software. The process of program design as a transition from a problem to a program is studied. A classification of software metrics is developed with the aim of better structuring and optimization of the software fuzzy metric design. Processes of constructing new measures from existing ones often use aggregation operations. Here we study aggregation operations for fuzzy set based software metrics.

Keywords: uncertainty, fuzzy set, problem, fuzzy software metric, software engineering

1 INTRODUCTION

Dealing with complex systems, we frequently encounter situations with incomplete, vague, and uncertain knowledge. To study, build, and utilize such systems, we need definite knowledge about this imprecision and incompleteness. This knowledge is obtained through measurement.

Now software systems are at top complexity levels. That is why, it is so important to study fuzzyness and imprecision in software engineering. Software measurement plays an important, and sometimes critical, role on all stages of the software life cycle. Taking the design stage, we see that software projects are notorious for running over schedule and budget, yet still having quality problems [24]. To improve design processes software metrics are elaborated and utilized. Such metrics allow the manager to quantify the schedule, work effort, product size, project status, and quality performance. If the current performance is not measured and the data are not used to improve future work estimates, those estimates will just be guesses. Metrics help one to better control software projects and learn more about the project organization. For instance, a project manager cannot quantify how well new development processes are working without some measure of current performance and a baseline to compare against.

The area of software measurement, is one of the areas in software engineering where researchers are active since more than thirty years. The main tool area in this area is software metrics. A metric is here not considered in the sense of a mathematical metric space, but is treated as a quantitative property of programs and related processes obtained in measurement.

A software metric is traditionally built as a mapping from a software object to a set of numerical values in order to quantify a software attribute. One goal for such mappings is to assess the quality of targeted software attributes such as maintainability, efficiency, clarity, etc.

At the same time, as Ruhe [25] writes, analysis of software engineering data is often concerned with treatment of incomplete knowledge, with management of inconsistent pieces of information and with manipulation of various levels of representation of data. Existing techniques of data analysis, mainly based on quite strong assumptions (some knowledge about dependencies, probability distributions, large number of experiments), are unable to derive conclusions from incomplete knowledge, or cannot manage inconsistent pieces of information. Even when knowledge is complete and consistent, it is impossible to achieve full certainty in projected and predicted characteristics. That is why fuzzy software metrics have been created and used for different purposes [2, 3, 5, 19, 20, 21, 26].

The goal of this paper is to develop a system approach to software measurement, reflecting uncertainty, imprecision, and incompleteness in evaluation of software and related processes. To achieve this goal, we consider principles of software metric design in Section 2. In Section 3, a classification of software metrics is suggested that develops classifications from [12, 22]. In Section 4, we investigate how fuzziness in software measuring emerges and how to take this fuzziness into account. In Section 5, we study aggregation operations for fuzzy sets used for building efficient software metrics and their relevant utilization in software engineering.

2 MEASURING SOFTWARE QUALITY

As for any other engineering discipline, measurement is pivotal for software engineering. Software metrics are measures used to quantify software, software development resources, and/or the software development process. This includes items that are directly measurable, such as *lines of code*, as well as values that are calculated from measurements, such as *earned value*. In all cases, a metrics has numerical values. At the same time, as Prather writes [24], “it is fairly certain that no one “magic number” can serve as a measurement for all

characteristics of software that might be considered important…"

This conclusion is supported not only by empirical data and history of software, but also by results of the general theory of evaluation [15] as the process of evaluation has three main stages: *preparation*, *realization*, and *analysis*.

Preparation demands the following operations to achieve correct and sufficiently exact evaluation:

1. Choosing evaluation criteria.
2. Corresponding characteristics (indices) to each of the chosen criteria.
3. Representing characteristics by indicators (estimates).

This shows that a complete process of evaluation preparation has the following structure:

Criterion → Index → Indicator

A specific realization of this process is the GQM (Goal-Question-Measurement) approach to software measurement [1]. Creation of software metrics has to include the following three stages:

1. Setting goals specific to needs in terms of purpose, perspective, and environment.
2. Refinement of goals into quantifiable tractable questions.
3. Deducing metric and data to be collected (as well as the means for their collection) to answer the questions.

Thus, the first stage in evaluation demands to determine a specific criterion for evaluation. This criterion gives the goal of evaluation. Criteria of good software include such properties as reliability, adequacy, exactness, completeness, convenience, user friendliness etc. However, such properties are also directly immeasurable and to estimate them, it is necessary to use corresponding indicators or indices. With respect to software quality such indicators are called software metrics. However, a criterion can be too general for direct estimation. This causes necessity to introduce more specific properties of the evaluated object. To get these properties, quantifiable tractable questions are formulated. Such properties play the role of indices for this criterion. Thus, the second stage of evaluation consists of index selection that reflects criteria. Sometimes an index can coincide with the corresponding criterion, or a criterion can be one of its indices. However, in many cases, it is impossible to obtain exact values for the chosen indices. For instance, we cannot do measurement with absolute precision. What is possible to do is only to get some estimates of indices. Consequently, the third stage includes obtaining estimates or indicators for selected indices. In the case of software, these indicators have form of software measures.

Similar approach was suggested by Belchior, Xexéo, and da Rocha [3] in their hierarchical software quality evaluation model (SQEM). This model is based on four main concepts: objectives or goals, factors, criteria and evaluation processes. Quality objectives or goals form criteria and represent important properties that a product should possess. Each goal is decomposed in factors, which can be or not further decomposed in subfactors. Factors and subfactors play the role of indices and define different users' perspectives about the quality of a software product.

However, a software metric is useful only when there are corresponding procedures/algorithms of measurements.Thus we need more stages for metric development.

4. Designing procedures/algorithms for data collection
5. Designing procedures/algorithms for computing metrics values.
6. Designing procedures/algorithms for analysis of measurement results.

It is necessary to remark that measuring algorithms are usually recursive, while programs are dynamic objects that are during their life cycle. To facilitate this process, many software update services are suggested on the Web. That is why, super-recursive algorithms [7, 8] would be more efficient and flexible for measurements in software engineering.

3 SOFTWARE METRIC TYPOLOGY

Here we use a general concept of a software metric given in [14]. Let **A** be a class of programs or software related processes/systems and *L* be a partially ordered set.

Definition 1. Any function **m**: **A** → *L*, or **n**: I → *L*, is an *internal software metric* or *measure* with the scale *L*.

This allows us to consider three types of fuzzy set software metrics: metrics that have elements of fuzzy set as their values, metrics that have fuzzy set as their values, and metrics that have linguistic elements as their values,By relation of measurement procedures to software, there are three types of metrics:

1. *Direct software metrics* represent properties of software.
2. *Related software metrics* represent properties of systems and processes related to software.
3. *Intermediate software metrics* represent properties of relations between software and other systems and processes.

LOC, SLOC, and the cyclomatic number are direct software metrics. Progress metrics are related software metrics. Trouble report metrics are intermediate software metrics.

Any program is aimed at solution of some problem. So, to measure quality of the future program and its design, it is necessary to relate measurement to the corresponding problem.

A problem consists of three parts: absence of some object, understanding of this absence, and a feeling of a need for this object. Such absent object may be some information, for example, what weather will be tomorrow, or some physical object such as a house or car.

Having a problem, we want to solve it, to get a solution. At first, we consider *static solutions*.

Definition 2. a) A *final solution* to a problem is an object for which there is a need in the problem. b) An *intermediate solution* to a problem is a system that will be able to give us a final solution when necessary. c) A *start solution* to a problem is a system that will give us an intermediate solution.

For instance, biologists want to know the structure of genes in an organism. This is the principal problem of bioinformatics now. The final solution of this problem is a description of the gene structure. However, genetic structures are very complex and biologists cannot discover these structures without help of computers. Thus, an intermediate solution to this problem is a computer program that helps to discover the gene structure given information of corresponding biological measurements. A start solution to this problem is a program designer or a team of program designers who will write such a program.

Definition 3. A *dynamic* (start, intermediate, or final) *solution* to a problem is a process that gives as its result a static (start, intermediate, or final) solution to this problem.

Taking our example, we see that a dynamic start solution is a process of finding or organization of a team of program designers, a dynamic intermediate solution is a process of designing/writing the necessary program, and a dynamic final solution is a computational process that gives the necessary genetic structure.

The triadic model of a problem solution allows us to classify software metrics with respect to problems of software design, maintenance, and utilization. In these problems, software is considered as a final solution. It gives us three static (object, project, and organization) metrics and three dynamic (object process, project process, and organization process) metrics.

Definition 4. a) The *object software metric* represents (reflects) properties of software. b) The *project software metric* represents (reflects) properties of software-oriented projects. c) The *organization software metric* represents (reflects) properties of software-oriented project organization.

An example of an object software metric is software safety. An example of a project software metric is the cost of software design. An example of an organization software metric is the Capability Maturity Model for Software (CMM) aimed at assisting organizations in improving their software process [19].

Kan [18] introduces three classes of software metrics:

1. Product software metrics describe the characteristics of the product.
2. Process software metrics can be used to improve software development and maintenance.
3. Project software metrics describe the project characteristics and execution.

A classification by the object of measurement:

1. *Textual software metrics* are determined directly by characteristics of the text of a program.
2. *Model software metrics* are determined by models of a program.
3. *External software metrics* are determined by external factors such as cost or the number of programmers.

4 FUZZINESS IN SOFTWARE MEASURING

Fuzziness in knowledge of some system emerges from three sources: people's ignorance, intrinsic properties of the system itself, and intrinsic properties of information retrieval process. Mathematical technique allows one to represent ignorance of people and artificial intelligent systems, vagueness of information, and boundaries of knowledge in an exact way. According to Bonissone and Tong [4], ignorance is subdivided in to three large categories:

Incompleteness covers cases where some data (e.g., the value of a variable) are missing (unknown).

Imprecision covers cases where some data (e.g., the value of a variable) are given but not with the precision required.

Uncertainty covers cases where an agent is not certain on some given data (e.g., the value of a variable).

Properties of systems that lead to fuzziness are graduality, as well as incompleteness and partiality of the property.

Thus, fuzzy set software metrics must reflect:

1. Uncertainty in:
   - 1.a. Initial data (input information for measurement);
   - 1.b. Functioning of the measurement algorithm;
   - 1.c. Intrinsic properties of measured objects.
2. Imprecision in:
   - 2.a. Initial data (input information for measurement);
   - 2.b. The measurement algorithm (the function defined by it can be fuzzy);
   - 2.c. The measurement algorithm (the function defined by it can be crisp, but allows one to get only approximate values).
3. Incompleteness in:
   - 3.a. Initial data (input information for measurement);
   - 3.b. The measurement algorithm as it can give incomplete result;
   - 3.c. The measurement algorithm as it can use not all necessary data and reflect not all relevant aspects/parameters of the measured object.
4. Gradation of a software property.
5. Incompleteness of a software property (at a given time).
6. Partiality of a software property.

For instance, when the measure SLOC or LOC is evaluated at the beginning of the software development process, there is uncertainty due to the lack of knowledge and it is possible to give only some estimations for the real value of SLOC (LOC). That is why in this case, it is more reasonable to use a fuzzy version of SLOC (LOC), in which the certainty of experts is represented explicitly.

Functionality of a program P with respect to functions that are realized by P is a graded property. The simple functional grade of P is the number of realized functions. For instance, if P realizes functions f_1 ,…, f_n , then the *simple functionality* of P is n. The weighted *functionality* of P is equal to $\Sigma_{i=1}^{n} a_i$ where a_i is the weight of the function f_i.Weights reflect importance of functions. Cost is another example of a graded property.

The achieved level in software system development is an example of an incomplete property. For instance, if we define *completeness of a software design project* as the part of software that has been already coded, then we see that completeness is naturally a graded property and we represent it by a fuzzy set with one element. Moreover, contemporary software is a complex multicomponent

system. Consequently, we will have more information on the project development if we assign *design completeness* to each of the components. This gives us a fuzzy set, elements of which are software components and the membership function indicates to what extent this component is completed.

It is useful also to consider such a software property as *completeness of software system validation*. This property is also necessary for hardware design where it becomes especially important.

An example of a partial property is program *correctness*. The most common measurement for correctness is defects per KLOC (KLOC = Thousand Lines of Code), where a defect is defined as a verified lack of conformance to the requirements. A user of the program reports defects after the program has been released for general use.

Definition 5 [29]. A *fuzzy set* A in a set U is the triad $(U, \mu_A, [0,1])$, where μ_A: $U \to [0,1]$ is a *membership function* of A and $\mu_A(x)$ is the *degree of membership* in A of $x \in U$.

If A is a nonfuzzy (crisp) set, then μ_A takes only two values: 0 and 1. Fuzzy sets are special cases of named sets [6, 9].

5 AGGREGATION OPERATIONS IN FUZZY SET SOFTWARE MEASUREMENT

Aggregation operations on fuzzy sets are operations by which several fuzzy sets are combined in a desirable way to produce a single fuzzy set. For instance, in [21] the authors use max-min aggregation and sum-product aggregation to define the distance between to projects of software design.

Let us consider some examples of aggregation in software measurement. As Demko, Pizzi, and Somorjai write [17], one reason for software measurement is to assess the quality of targeted software attributes such as maintainability, extensibility, efficiency, and clarity. In essence, this is a problem of classification: given a set objects with known features (software metrics) and group labels (quality rankings), design a classifier that can predict the group characteristics of new objects using only the software metrics. For reliability, a panel of experts (system architects) usually determines the group characteristics indicating quality. To combine all produced estimates into an integral characteristic used to predict object quality, aggregation of fuzzy sets is utilized.

An aggregation operation on n fuzzy sets ($n \geq 2$) is defined by the following rule based on an *aggregation function* f: $[0,1]^n \to [0,1]$:

If $A = (U, \mu_A, [0,1])$ is the result of aggregation of fuzzy sets $A_1 = (U, \mu_{A1}, [0,1])$, … , $A_n = (U, \mu_{An}, [0,1])$, then $\mu_A(a) = f(\mu_{A1}(a), \dots, \mu_{An}(a))$ for all a from U.

Example 1. Standard operations with fuzzy sets give examples of aggregation. For two fuzzy sets $A_1 = (U, \mu_{A1}, [0,1])$ and $A_2 = (U, \mu_{A2}, [0,1])$, the main operations are:

The intersection $C = (U, \mu_C, [0,1]) = A_1 \cap A_2$ where $\mu_C(a) = \min\{\mu_{A1}(a), \mu_{A2}(a)\}$ and the aggregation function $f(x, y) = \min(x, y)$;

The union $E = (U, \mu_E, [0,1]) = A_1 \cup A_2$ where $\mu_E(a) = \max\{\mu_{A1}(a), \mu_{A2}(a)\}$ and the aggregation function $f(x, y) = \max(x, y)$;

The average $A = (U, \mu_A, [0,1]) = \text{av}(A_1, A_2)$ where $\mu_E(a) = ½(\mu_{A1}(a) + \mu_{A2}(a))$ and the aggregation function $f(x, y) = ½(x + y)$;

Aggregation of fuzzy sets satisfy the following axioms:

Axiom A1. The *boundary condition*: $f(0, 0, ..., 0) = 0$ and $f(1, 1, ..., 1) = 1$.

Axiom A2. *Monotonicity*:

For any pair $\langle a_1, a_2, ..., an\rangle$ and $\langle b_1, b_2, ..., b_n\rangle$ of n-tuples with $a_i, b_i \in [0,1]$, if $a_i \leq i$ for all $i = 1, \dots, n$, then $f(a_1, a_2, ..., a_n) \leq f(b_1, b2, ..., b_n)$.

Axiom A3. *Continuity*: f is a continuous function.

We call this operation *pointwise aggregation* and consider two other aggregation operations: graded and collected aggregation.

Results of *graded aggregation* depend on the types of aggregated fuzzy sets. Let us assume that aggregation operation is applied only to fuzzy sets that have types $t1, t2, \dots, tk$. For a fuzzy sets $A = (U, \mu_A, [0,1])$, $t(A)$ denotes the type of A. Then a graded aggregation operation on n fuzzy sets ($n \geq 2$) is defined by the following rule based on a system of aggregation functions $f_{t1,\dots,tn} : [0,1]^n \to [0,1]$:

If $A = (U, \mu_A, [0,1])$ is the result of aggregation of fuzzy sets $A_1 = (U, \mu_{A1}, [0,1])$, … , $A_n = (U, \mu_{An}, [0,1])$, then $\mu_A(a) = f_{t(A1),\dots,t(An)}(\mu_{A1}(a), \dots, \mu_{An}(a))$ for all a from U.

Example 2. It is possible to take such an aggregation function f that gives an "average" or "typical" value of its arguments, or in other words, f is a measure of central tendency. Usually the mean, which is the arithmetic average of the data distribution, is used as the preferred measure of central tendency. However, there are cases when the mean is not the "best" measure of central tendency. In certain situations, the median, which is the midpoint of the data distribution, is the preferred measure: when the distribution is skewed or there is a small number of data. The purpose for reporting the median in these situations is to combat the effect of *outliers.* Outliers affect the distribution because they are extreme scores. Thus, we take different functions f for aggregation depending on properties of aggregated fuzzy sets.

Pointwise aggregation of fuzzy sets is a special case of graded aggregation when all fuzzy sets have one type.

In *collected aggregation* not only membership function values are aggregated, but also objects in the universe U of discourse. It involves three functions: an aggregation function f in [0, 1], aggregation function g in U, and integral operation Q in [0,1] (cf. [10]), where f: $[0,1]^n \to [0,1]$, g: $U^n \to U$, and Q: $\cup_{i=1}^{\infty} [0,1]^i \to [0,1]$. Then a collected aggregation operation is defined as:

If $A = (U, \mu_A, [0,1])$ is the result of collected aggregation of fuzzy sets $A_1 = (U, \mu_{A1}, [0,1])$, … , $A_n = (U, \mu_{An}, [0,1])$, then $\mu_A(a) = Q\{f(\mu_{A1}(a_{i1}), \dots, \mu_{An}(a_{in})); g(a_{i1}, \dots, a_{in}) = a\}$ for all a from U.

Example 3. Operations with fuzzy numbers give examples of collected aggregation. For two fuzzy numbers $A_1 = (R, \mu_{A1}, [0,1])$ and $A_2 = (R, \mu_{A2}, [0,1])$, the main operations are:

Addition $C = (R, \mu_C, [0,1]) = A_1 + A_2$ where $\mu_C(a) = \sup_{x+y=a} \min\{\mu_{A1}(x), \mu_{A2}(y)\}$, the aggregation function $f(x, y) = \min(x, y)$, $g(x, y) = x + y$, and $Q(a_i) = \sup\{a_i\}$;

Multiplication $E = (U, \mu_E, [0,1]) = A_1 \cdot A_2$ where $\mu_E(a) = \sup_{x \cdot y=a} \min\{\mu_{A1}(x), \mu_{A2}(y)\}$, the aggregation function $f(x, y) = \min(x, y)$, $g(x, y) = x \cdot y$, and $Q(a_i) = \sup\{a_i\}$.

Definition 6. An integral operation Q is called monotone (antitone) if the following statement is true

$$\forall\, n, i\ (x_i > y_i) \Rightarrow (\omega(x_{1,\dots,} x_{i-1}, x_i, x_{i+1}, \dots, x_n) > Q(x_1, \dots, x_{i-1}, y_i, x_{i+1}, \dots, x_n))$$

$$(\forall\, n, i\ (x_i > y_i) \Rightarrow (\omega(x_{1,\dots,} x_{i-1}, x_i, x_{i+1}, \dots, x_n) < Q(x_1, \dots, x_{i-1}, y_i, x_{i+1}, \dots, x_n)))$$

Let us define a function g: $U^n \to U$ by the following conditions: $g(a, \dots, a) = a$ for all a from U, and g is undefined for all other arguments. By the definition of an integral operation [10], we have $\mu_A(a) = Q\{f(\mu_{A1}(a_{i1}), \dots, \mu_{An}(a_{in}))$; $g(a_{i1}, \dots, a_{in}) = a\} = f(\mu_{A1}(a), \dots, \mu_{An}(a))$ for all a from U. This gives us the following result.

Proposition 1. Pointwise aggregation of fuzzy sets is a special case of collected aggregation.

Usually software metrics satisfy some additional conditions and have additional properties (cf., [24, 27, 18, 16, 12, 13]). Let us consider some of them.

One of the most important is program monotonicity introduced in [24].

Definition 7. A software metric m is called *program monotone* if $m(Q) \geq m(P)$ for any program Q and any its subprogram P.

Remark 1. Program monotonicity is a principal property of software metrics because it is natural to suggest that a part is always simpler, or has lower complexity, than the whole. However, not all software metrics are monotonous. For instance, if we take such software metric as reliability or safety, their values are usually higher for subprograms than for the whole programs.

Thus, it is natural to consider such a property as program antitonicity.

Definition 8. A software metric m is called *program antitone* if $m(P) \geq m(Q)$ for any program Q and any its subprogram P.

Usually cost and time of the development are program monotone. However, component-based software development changes these properties and they become antitone. Really, if we use some existing program P as a component of a program Q, cost of Q and time for its development can be less than cost of P and time for its development.

Aggregation is used to build software metrics. So, it is useful to know when this operation preserves monotonicity.

Theorem 1. If the aggregation function f is monotone (antitone), then pointwise aggregation preserves monotonicity (antitonicity).

Theorem 2. If all aggregation functions $f_{t1,\dots,tn}$ are monotone (antitone), then graded aggregation preserves monotonicity (antitonicity).

Theorem 3. If the aggregation function f and integral operation Q are monotone (antitone), then collected aggregation preserves monotonicity (antitonicity).

Let h be a quantitative (ordered) characteristic of the design process or/and its product – a software system p.

Definition 9. The characteristic h is called *amplifying* if the larger value of this characteristic implies/reflects the higher quality of the process/product. Metrics reflecting such a characteristic are also called *amplifying*.

The number of realized functions, reliability, security, and the number of completed tasks in the process of development are amplifying characteristics.

Definition 10. The characteristic h is called *curtailing* if the smaller value of this characteristic implies/reflects the higher quality of the process/product. Metrics reflecting such a characteristic are also called *curtailing*.

The number of bugs in a program, cost of the development, and the number of trouble reports in the process of development are curtailing characteristics.

Definition 11. The characteristic h is called *stabilizing* if it has to stay stable during the process. Metrics reflecting such a characteristic are also called *stabilizing*.

Theorem 4. a) If $g_1, g_2, \dots, g_n$ are amplifying metrics and the aggregation function f is monotone (antitone), then their pointwise aggregation gives an amplifying (curtailing) metric.

b) If $g_1, g_2, \dots, g_n$ are curtailing metrics and the aggregation function f is monotone (antitone), then their pointwise aggregation gives a curtailing (amplifying) metric.

Theorem 5. a) If $g_1, g_2, \dots, g_n$ are amplifying metrics and all aggregation functions $f_{t1,\dots,tn}$ are monotone (antitone), then their graded aggregation gives an amplifying (curtailing) metric.

b) If $g_1, g_2, \dots, g_n$ are curtailing metrics and all aggregation functions $f_{t1,\dots,tn}$ are monotone (antitone), then their graded aggregation gives a curtailing (amplifying) metric.

Theorem 6. a) If $g_1, g_2, \dots, g_n$ are amplifying metrics, the aggregation function f and integral operation Q are monotone (antitone), then their collected aggregation gives an amplifying (curtailing) metric.

b) If $g_1, g_2, \dots, g_n$ are curtailing metrics the aggregation function f and integral operation Q are monotone (antitone), then their collected aggregation gives a curtailing (amplifying) metric.

6 CONCLUSION

In the paper, inherent fuzziness in software engineering is studied. A methodology is developed for building software metrics with values in fuzzy sets and linguistic variables. This allows programmers and project managers to reflect ignorance of people and artificial intelligent systems, vagueness of information, and boundaries of knowledge in an exact way. At the same time, as it is demonstrated in [11, 14], it is necessary to build and use vector valued and higher dimensional software metrics. It would be interesting to extend fuzziness to vector valued and higher dimensional software metrics considered in [16].

Another direction for a research is a study and design of such software metrics in which uncertainty, imprecision, and incompleteness are represented by rough sets. An example of such metrics is considered in [25].

One more direction for a research emerges with the advent of super-recursive programming [8, 10]. Properties of such programs and algorithms on which they are based indicate that these programs have more fuzzy characteristics than conventional recursive programs and it would be necessary to develop specific metrics for such programs.

7 REFERENCES

[1] V. R. Basil and H.D. Rombach, The TAME project: Towards Improvement-Oriented Software Evironments, *IEEE Transactions on Software Engineering*, v. 14, No. 6, pp. 758-773, 1988

[2] F. B. Bastani, DiMarco, G., Pasquini, A. and Brancati, V. Experimental evaluation of a fuzzy-set based measure of software correctness using program mutation, *Proceedings of the 15th Int. Conf. on Software Engineering*, Baltimore, Maryland, pp. 45 - 54

[3] A.D. Belchior, Xexéo, G. and da Rocha, A.R.C. Evaluating Software Quality Requirements using Fuzzy Theory, *Proceedings of ISAS* 96, Orlando, 1996

[4] Bonissone, P.P. and Tong, R. Reasoning with uncertainty in expert systems, *Man Machine Studies*, v. 22, pp. 241-150, 1985

[5] M. R. Braz, and S. R. Vergilio, Using Fuzzy Theory for Effort Estimation of Object-Oriented Software, *16th IEEE International Conference on Tools with Artificial Intelligence* (ICTAI'04) pp. 196-201, 2004

[6] M. Burgin, Theory of Named Sets as a Foundational Basis for Mathematics, in "*Structures in Mathematical Theories*", San Sebastian, pp. 417-420, 1990

[7] M. Burgin, How We Know What Technology Can Do, *Communications of the ACM*, v. 44, No. 11, pp. 82-88, 2001

[8] M. Burgin Algorithmic Complexity of Recursive and Inductive Algorithms, *Theoretical Computer Science*, v. 317, No. 1/3, 2004, pp. 31-60, 2004

[9] M. Burgin, *Unified Foundations of Mathematics*, Preprint in Mathematics LO/0403186, 2004 (http://arXiv.org)

[10] M. Burgin, *Superrecursive Algorithms*, Springer, New York, 2005

[11] M.S. Burgin, and Chelovskii, Yu. A. Multidimensional structural analysis of FORTRAN programs, *Control Systems and Machines*, No. 1, pp. 38-39, 1982 (in Russian)

[12] M.Burgin, and Debnath, N.C. "Complexity of Algorithms and Software Metrics", in Proc. ISCA 18th Int. Conf. "*Computers and their Applications*", Honolulu, pp. 259-262, 2003

[13] M.Burgin, and Debnath, N.C. "Hardship of Program Utilization and User-Friendly Software", in *Proc. ISCA* 16th *Int. Conf.* "*Computer Applications in Industry and Engineering*", Las Vegas, pp. 314-317, 2003

[14] M. Burgin, and Debnath, N.C. "Measuring Software Maintenance", in Proceedings of the ISCA 19th International Conference "*Computers and their Applications*", ISCA, Seattle, Washington, pp. 118-121, 2004

[15] M. Burgin, and Kavunenko, L. *Measurement and Evaluation in Science*, Kiev, 1994 (in Russian)

[16] M. Burgin, H. K. Lee and N. Debnath, *Software Technological Roles, Usability, and Reusability*, in Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IEEE Systems, Man, and Cybernetics Society, Las Vegas, pp. 210-214, 2004

[17] A.B. Demko, Pizzi, N.J., and Somorjai R.L. Scopira - A system for the analysis of biomedical data. *Proc IEEE Canadian Conference on Electrical and Computer Engineering*, Winnipeg, Canada, May 12-15, 1093-1098, 2002

[18] P.T. Devanbu, Karstu, S., Melo, W., and Thomas, W. "Analytical and Empirical Evaluation of Software Reuse Metrics", *IEEE Proceedings of ICSE*-18, pp. 189-199, 1996

[19] N. E. Fenton, P. Krause, M. Neil, Software Measurement: Uncertainty and Causal Modeling, *Software*, V. 19, No. 4, pp. 116-122, 2002

[20] A. Idri, and Abran, A. A Fuzzy Logic Based Set of Measures for Software Project Similarity: Validation and Possible Improvements, *Seventh International Software Metrics Symposium*, London, England, pp. 85-96, 2001

[21] A. Idri, Abran, A. and Khosgoftaar, T.M., Fuzzy Analogy: A New Approach for Software Cost Estimation, *Int. Workshop on Software Measurement* (IWSM'01), Montréal, Québec , pp. 93-101, 2001

[22] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd Edition, Addison Wesley Professional, 2002

[23] M. Paulk, Weber, C.V., and Curtis, B. *The Capability Maturity Model for Software guidelines for improving the software process*, Addison Wesley, 1994

[24] R.E. Prather, An Axiomatic Theory of Software Complexity Measure, *Computer Journal*, v. 27, No. 4, 340-347, 1984

[25] G. Ruhe, Rough Set Based Data Analysis in Goal Oriented Software Measurement, 3rd International Software Metrics Symposium (METRICS '96) "*From Measurement to Empirical Results*", Berlin, Germany, p. 10, 1996

[26] Sicilia, M.A., Cuadrado, J.J., Crespo, J. and García-Barriocanal, E. Software cost estimation with fuzzy inputs: fuzzy modeling and aggregation of cost drivers, *Kybernetika* 41, 249-264, 2005.

[27] E.J. Weyker, Evaluating Software Complexity Metrics, *IEEE Transactions on Software Engineering*, v. 14, No. 9, 1988, pp. 1357-1365

[28] K.E. Wiegers, *Creating a Software Engineering Culture*, Dorset House, 1996

[29] H.J. Zimmermann, *Fuzzy Set Theory and its Applications*, 2nd Ed., Kluwer, Dordrecht-Boston, 1991