Computational Intelligence Approaches for Software Quality Improvement

Grigore Albeanu, Henrik Madsen, and Florin Popențiu-Vlădicescu

Abstract Obtaining reliable, secure and efficient software under optimal resource allocation is an important objective of software engineering science. This work investigates the usage of classical and recent development paradigms of computational intelligence (CI) to fulfill this objective. The main software engineering steps asking for CI tools are: product requirements analysis and precise software specification development, short time development by evolving automatic programming and pattern test generation, increasing dependability by specific design, minimizing software cost by predictive techniques, and optimal maintenance plans. The tasks solved by CI are related to classification, searching, optimization, and prediction. The following CI paradigms were found useful to help software engineers: fuzzy and intuitionistic fuzzy thinking over sets and numbers, nature inspired techniques for searching and optimization, bio inspired strategies for generating scenarios according to genetic algorithms, genetic programming, and immune algorithms. Neutrosophic computational models can help software management when working with imprecise data.

Keywords Computational intelligence · Immune algorithms · Software quality · Software reliability · Neutrosophic computational models

G. Albeanu
"Spiru Haret" University, Bucharest, Romania
e-mail: g.albeanu.mi@spiruharet.ro

H. Madsen
Danish Technical University, Lyngby, Denmark
e-mail: hmad@dtu.dk

F. Popențiu-Vlădicescu (✉)
University Politehnica of Bucharest & Academy of Romanian Scientists, Bucharest, Romania
e-mail: Fl.Popentiu@city.ac.uk

H. Pham (ed.), *Reliability and Statistical Computing*, Springer Series in Reliability Engineering, https://doi.org/10.1007/978-3-030-43412-0_18

1 Introduction

Improving quality of software is an important objective of any software developer. According to Jones [12], "software needs a careful analysis of economic factors and much better quality control than is normally accomplished".

The mentioned author has realized a deep analysis on software measurement and found out seven key metrics to be explored in order to estimate software economics and software quality with high precision [12]: the total number of function points, hours per function point for the project, software defect potential computed using function points, defect removal efficiency, delivered defects per function point; high-severity defects per function point; and security flaws per function point. For instance, the software defect potential is given by pow (number of function points, 1.25) [13], while the number of test cases can be estimated by pow (number of function points, 1.20), where pow (x, α) is x^{α}.

Many scientists, including Fenton and Pfleeger [9], Aguilar-Ruiz et al. [1], and Wójcicki and Dabrowski [38], have investigated software quality measurement using various metrics, statistical inference, and soft-computing methods.

Based on work [8] and the new developments in using artificial intelligence in software engineering [6, 22, 30, 33], this work considers classic and recent methods of Computational Intelligence (CI) applied in Software Engineering (SE) in order to identify improvements for solving the following tasks: software requirements analysis and precise software specification development; software development time reducing by evolving automatic programming and pattern test generation; dependability increasing by specific software design; and software cost/effort minimization by predictive techniques and optimal maintenance plans.

From the large variety of software quality definitions, in the following, the definition proposed by Jones [12] is used: "software quality is the absence of defects which would either cause the application to stop working, or cause it to produce incorrect results". As, software engineers proceed to develop a project, the main phases of the software life cycle are covered in this chapter.

The aim of this material, as an extension of [29], is to propose an extended approach based on fuzzy, intuitionistic fuzzy, and neutrosophic models for software requirement multi-expert evaluation (the third section), to update the usage of artificial immune algorithms for software testing (the fourth section), and to evaluate software reliability in neutrosophic frameworks (the fifth section).

2 Overview on Computational Intelligence Paradigms

According to IEEE Computational Intelligence Society [42], the main fields of Artificial Intelligence (AI) considered as special topics for CI are: Artificial Neural Networks (ANN), Fuzzy Systems (FS), Evolutionary Computation (EC), Cognitive and Developmental Systems (CDS), and Adaptive Dynamic Programming and Reinforcement Learning (ADP&RL). Coverage of the main paradigms can be found in [16], while innovative CI algorithms are presented in [39]. The impact of computational intelligence on software engineering developments was revealed in [26].

Inspired by the biological network of neurons, Artificial Neural Networks are used as nonlinear models to classify data or to solve input-output relations [16]. Based on a weighted directed graph, having three types of vertices (neurons)—input, hidden and output, the ANN makes use of three functions for every vertex: network input f_{in}, neuron activation f_{act}, and output f_{out}. If the associated graph is acyclic then ANN is a *feed forward* network, otherwise is a *recurrent* network. The weights of the network are obtained by a training process.

One kind of learning task, called *fixed*, uses a set of pairs—*training patterns*—(x, y), where x is an input vector, and y is the output vector produced by ANN when received as input the vector x. Both x and y can be multivariate with different dimensions. The learning process is evaluated by some metric, like square root of deviations of actual results from desired output.

Another kind of learning task, called *free*, use only input vectors, the objective of ANN addressing a *clustering/classification* requirement. Here, a similarity measure is necessary to identify the prototypes. The power of ANN depends on the activation model of neurons, and the number of hidden layers. It is well known the results [16]: "any Riemann-integrable function can be approximated with arbitrary accuracy by a multilayer perceptron". Both practical and theoretical results on using different types of ANN have increased the confidence in using ANN as computational intelligence models.

For software engineering, the following references used ANN to optimize the software development process: Dawson [7] and Madsen et al. [18].

When the inputs are fuzzy [40, 41], intuitionistic fuzzy [3, 17], or of neutrosophic type [31], the activation process is based on defuzzification/deneutrofication procedures. Fuzzy systems make use of fuzzy sets, fuzzy numbers, and fuzzy logic. An intelligent FS is a knowledge based system used to answer to questions/queries formulated by a user according to a linguistic variables language. The natural language processing based interface is responsible on fuzzification/neutrofication procedure. Neutrosophic thinking for engineering applications is based on three indicators: one for *truth/membership* degree (T), one for the degree of *indeterminacy* (I), and one for *false/non-membership* degree (F). If $F + T = 1$, and $I = 0$, then the fuzzy framework [40] is considered. If $F + T < 1$, and $I = 1 - (F + T)$, then the intuitionistic-fuzzy theory of Atanassov is obtained [3]. The neutrosophic framework considers $0 \leq T + T + F \leq 3$, with $0 \leq T, I, F \leq 1$. Defuzzification can be obtained easy by the centroid method. However, many other methods were proposed and used in various contexts. Converting an intuitionistic-fuzzy entity, or a neutrosophic entity to a crisp value is not so easy. Firstly, an *indicator function* estimating the holistic degree of truth/membership should be computed (as in fuzzy representation), and this function will be used to compute a crisp value. The indicator function, denoted by H, is computed by

$$H = \alpha T + \beta(1 - F) + \gamma I/2 + \delta(1 - I/2),$$

for every item from universe of discourse. The parameters α, β, γ, δ are positive numbers, in decreasing order, with their sum being 1. The method was proposed by Wang et al. [36], and the parameters should be found by the researcher based on the available information about the problem under treatment.

Other neutrosophic computational models are presented and used in the fifth section.

FS systems use tolerance for imprecision, uncertainty and partial truth to achieve tractability, robustness, low solution cost, and better rapport with reality, as Zadeh [42] recommended when he has coined the soft computing term.

Evolutionary computation is an area of research covering genetic algorithms, evolutionary strategies, and genetic programming. The techniques are based on a population of individuals and the following operations: *reproduction* (crossover/recombination), *random variation* (mutation, hypermutation), *competition*, and *selection*. The objective of any evolutionary algorithm is to optimize the searching process in a robust and intelligent manner, as inspired by biological reproduction schemes. Relevant results in software engineering were obtained, to mention some contributions, by Aguilar-Ruiz et al. [1], Arcuri and Yao [2], Berndt et al. [5], McGraw et al. [21], Michael et al. [23], Patton et al. [25], and Wappler and Lammermann [37].

Cognitive and Developmental Systems have specific targets in artificial life modelling and computational neuroscience: agent based modelling, special architectures of artificial neural networks, computational algorithms based on evolutionary strategies borrowed from real bio life. For software engineering, agent based modelling is one of CDS applications. The orientation on symbolic processing (as results of neural computing, concept transformation, and linguistic description) is the main reason to study CDS for software engineering. Also, the usage of machine learning for solving software engineering optimization problems motivates the application of CDS, as proposed by Wójcicki and Dabrowski [38] and Venkataiah et al. [35].

Adaptive Dynamic Programming and Reinforcement Learning are used to solve problems related to optimal control through efficient learning strategies. ADP&RL has to provide optimal decisions through knowledge based systems by active learning.

ADP&RL can be used in software engineering for optimal decision making along the software lifecycle, as described by Sultanov and Hayes [34].

3 Computational Intelligence for Software Requirements Engineering

The advancement in CI oriented technologies proved value in various SE specific applications: (1) automatic transformation of Natural Language Requirements into Software Specification; (2) software architecture design; (3) software coding and testing; (4) software reliability optimization; (5) software project management.

When consider Natural Language Requirements, the software engineer has to deal with ambiguity of requirements, incomplete, vague or imprecise descriptions, or the interpretation of the requirements. Requirements Ambiguity Reviews should be implemented at early phases of software development to obtain the following advantages [43]: requirements improvement and software defects reduction; 100% test coverage in order to identify software bugs; learn to differentiate between poor and testable requirements.

As Kamsties identified in [14], the requirements ambiguity reviewer should differentiate between *linguistic ambiguity* (context independent, lexical, syntactic) and *software engineering ambiguity* (context dependent, domain knowledge). If a sentence has more than one syntax tree (*syntactic ambiguity*), or it can be translated into more than one logic expression (*semantic ambiguity*) then an ambiguous requirement is found. When an anaphora or a pronoun has more than one antecedent, then a referential ambiguity should be processed. The analysts have also to do with pragmatic ambiguity generated by the relationship between the meaning of a sentence and its appearance place.

Special interest concerns the words like*: all, for each, each, every, any, many, both, few, some*, which are related to a whole set, or individuals in an unsized universe. Translating into logic expressions of sentences based on connectives like *and, or, if and only if, if then, unless, but* should address the truth membership degree and (intuitionistic) fuzzy norms, co-norms and implications. When detecting words like *after, only, with*, pronouns (*this, that, those, it, which, he, she*), *usually, often, generally, typically, normally*, and *also*, additional care is necessary when eliminates ambiguity. Other triggers announcing possibly ambiguity are given by *under-specified terms* (category, data, knowledge, area, thing, people etc.), *vague terms* (appropriate, as soon as possible, flexible, minimal, user-friendly, to the maximum extent, highly versatile etc.), and *domain-specific terms* (intelligence, security, level, input, source etc.). Osman and Zaharin described, in [24], an automated approach based on text classification to deal with ambiguous requirements.

However, automated disambiguation is impossible because human understanding is required to establish the requirements validity. In this case, a *multi-expert* approach is necessary to evaluate the requirements against ambiguity. The analysis proceeds in similar way for all fuzzy, intuitionistic-fuzzy, and neutrosophic approaches. A feasible strategy follows the steps:

1. Input one requirement as sentence in Natural Language or an Informal Language. Identify the requirement class (exact classification): Functional Requirements (FR), Nonfunctional Requirements (NFR), Performance/Reliability (PR), Interfaces (IO), Design constraints (DC).
2. After all requirements are considered, build the ambiguity degrees for every requirements class MFR[i], MNFR[i], MPR[i], MIO[i], and MDC[i] by every linguistic expert i, $i = 1, 2, \ldots, m$. The size of each matrix is given by the number of requirements identified for specified class. Consider a defuzzification/deneutrofication indicator, and for every requirement establish the tuple (r_k,

type$_k$, $e_{1,k}$, $e_{2,k}$, …, $e_{m,k}$), where $e_{i,k}$ denotes the truth indicator function associated by expert i to requirement k.

3. Every requirement r_k, having $e_{i,k} \geq 0.5$ for at least one expert will be considered by software requirements engineer for lexical, syntactical, and semantical analysis in order to obtain a set of interpretations S_k. Contextualize every interpretation by a clear description.
4. Start the re-elicitation procedure against customer/client team, in order to establish the true software requirements.

One recent initiative in SRE is NaPIRE [10] which identifies the best practices used over the world. However, addressing the agile development paradigm, the practice already reported is not suited to other software development paradigms. The above proposal can be a setup for any paradigm, including the waterfall classic approach.

4 Computational Intelligence for Software Testing

There are a large variety of software testing methods and techniques which are different from the point of view of the paradigm type, effectiveness, ease of use, cost, and the need for automated tool support.

Evolutionary techniques can be applied to code development and for generating test cases, or unit testing. Also, evolutionary strategies apply when someone wants to estimate software development projects, as Aguilar-Ruiz et al. presented in [1].

The lifecycle of any evolutionary algorithm for software testing starts with a number of suitable test cases, as *initial population*. The evaluation metric is always based on a *fitness function*. Let be n the total number of domain regions/testing paths, k be the number of regions/paths covered by a test, hence the test case associated fitness/performance is k/n. The recombination and mutation operators applied to test case work as usual procedures applied to sequences/strings and are influenced only by the test case structure (representation).

Recently, Arcuri and Yao [2] use *co-evolutionary algorithms* for software development. In co-evolutionary algorithms, two or more populations co-evolve influencing each other, in a cooperative co-evolution manner (the populations work together to accomplish the same task), or in a competitive co-evolution approach (as predators and preys in nature).

Given a software specification S, the goal of software developer is to evolve a program P along some iterations in order to satisfy P. If genetic programming is used, the fitness of each program in a generation is evaluated on a finite set T of unit tests that depends on the specification S; the fitness value of a program being the maximum number of unit tests passed by the program. A better approach consists of using different sets T_i of unit tests for each new generation i.

The generation of new sets of unit tests can use the *negative selection* approach described in Popentiu and Albeanu [28]. If G_i is the set of genetic programs at step i, and T_i is the set of unit tests for the i-th generation, the next generation is obtained according to the following method.

Let g(t) be the output of the program g having input t. Let c(t, g(t)) the fitness of the output of g related to the true output, when the precondition is valid, otherwise c = 0. If N(g) is the number of vertices of the program g (as flowchart), and $E_i(g)$ is the number of errors generated by g on t in T_i, the fitness degree of g is [2]:

$$f(g) = N(g)/(N(g) + 1) + E_i(g)/(E_i(g) + 1) + \Sigma\{c(t,\ g(t));\ t \text{ in } T_i\},\ g \text{ in } G_i.$$

Clonal selection can be used to test generation (a large collection of test cases can be obtained by mutation operator). The size of collection, considered like detectors, can be reduced by simulating a negative selection to eliminate those detectors which are not able to detect faults. The remaining detectors will be cloned and mutated, evaluated and used to create a new population of detectors.

The framework proposed by Arcuri and Yao can benefit from new algorithms for test case generation based both on genetic [5, 21, 23, 25] and immune [28] algorithms.

According to [20], for software testing and debugging, other models can be used:

1. Let K be the number of fault classes established by an expert, D be the program input domain, partitioned in n (n > 1) regions D_1, D_2, …, D_n, and Q be the probability distribution giving the operational profile: $\Sigma\{Q(x): x \in D\} = 1$. If $\phi_k(x)$ is the membership degree of x to the kth fault domain, and $P(f_k)$ is the probability to experience a kth type fault, then every software run will succeed with the degree R given by

$$R = 1 - \sum_{i=1}^{n} \int_{D_i} (\max_{j=1,\ldots,K} P(f_j)\phi_j(x))Q(x)dx$$

   When the membership degrees φ_k are obtained by probability conversion, then R is the software reliability obtained in the probabilistic framework, according to Bastani and Pasquini [4]. Otherwise, R can be viewed as the membership degree when the universe of discourse contains all software items and the fuzzy set of the reliable items are considered.
2. Both the degree of detectability and the degree of risk can be estimated using fuzzy systems [20]. According to [11], the detectability of a test T is the probability that T is able to detect a bug in a selected software unit, if this software contains a bug. The degree of detectability of a testing approach T can be obtained by *mutation testing*. Also, the detectability can be given by a linguistic variable (*low, about low, average, about high, high*) with modifiers (hedges) like: *very, slightly, more-or-less* etc. In this way a situation like "the method T suspects a bug, more investigation are necessary" can be modelled and considered for a fuzzy rule database of an expert system for software testing.
3. Mutation is also a valuable operation used in the case of fuzz testing, or fuzzing [15]. According to [27], "fuzzing has long been established as a way to automate negative testing of software components". Mainly, fuzz testing is a technique for

software testing that generates random data to the inputs of a software unit in order to identify the existence of defects when the program fails.

5 A Neutrosophic Approach to Software Quality Evaluation

Neural networks and genetic algorithms can be used to provide an optimal reliability allocation in the case of modular design under fault tolerant constraints. These techniques were used by Madsen et al. [18, 19] in various contexts. In this section we use the neutrosophic numbers of Smarandache to evaluate the reliability/availability of software under a fault-tolerance design.

For our considerations, a *neutrosophic number* is an object of form $a + bI$, where a and b are real numbers, I is an operation such as $I^2 = I$, $I - I = 0$, $I + I = 2I$, $0I = 0$, with 1/I and I/I are not defined [31]. If $x = a + bI$ and $y = c + dI$ are neutrosophic numbers, then [32]:

(a) $x + y = (a + c) + (b + d)I$;
(b) $x - y = (a - c) + (b - d)I$;
(c) $xy = (ac) + (ad + bc + bd)I$;
(d) $\lambda x = (\lambda a) + (\lambda b)I$;
(e) $x/y = u + vI$ (when ever is possible), with $u = a/c$, and $v = (bc - ad)/(c^2 + cd)$.

In order to obtain u and v in (e), the following identification chain should be followed, according to the rules (a) and (c):

$$\begin{aligned}
&(u + vI)(c + dI) = (a + bI),\\
&uc + (ud + cv + vd)I = (a + bI),\\
&uc = a,\ u = a/c,\ ad/c + cv + vd = b,\ ad + c^2v + vcd = bc,\\
&v\left(c^2 + cd\right) = bc - ad
\end{aligned}$$

The reliability of parallel structures is given by $R(A, B) = 1 - (1 - A)(1 - B) = X + IY$, with A, and B given by neutrosophic numbers, and $1 = 1 + 0I$. Therefore, when the reliability is appreciated by human experts as indeterminate, the new calculus permits the computation of the reliability, and the result may be interpreted as *minimum value* X, the *median* $X + Y/2$, the *first quartile* $X + Y/4$, or the *third quartile* $X + 3Y/4$ depending on the *deneutrofication procedure*.

The reliability of serial connected modules is given by $R(A, B) = AB$ (according to the rule c from above). This methodology can be used to evaluate the reliability of the bottom-up structures, and to transform reliability allocation/optimization problems formulated in neutrosophic manner. The solving strategy follows similar steps as describing by Madsen [17], and Madsen [19].

Neutrosophic numbers can be used for a neutrosophic estimation of the maintenance effort, based on a neutrosophic variant of the model of Belady and Lehman (cited by [9]):

$$\mathrm{M} = \mathrm{p}_1 + \mathrm{K}(1 + \mathrm{d}_1 - \mathrm{f}_1) + \mathrm{I}\big(\mathrm{p}_2 + \mathrm{K}(1 + \mathrm{d}_2 - \mathrm{f}_2)\big],$$

where $p_1 + Ip_2$ is the productive effort that involve analysis, design, coding, testing, and evaluation, $d_1 + Id_2$ is a complexity measure associated with poor design, $f_1 + If_2$ is the maintenance team unfamiliarity with the software, and K is an empirical constant. For instance, if the development effort was 480 + 20I persons per month, the project complexity was 7 + 2I, the degree of familiarity is 0.7 + 0.1I, with K = 0.25, then M = 481.825 + 24.75I which after deneutrofication by median can give the total effort expended in maintenance as 494.2 persons per month.

If a software will be reused after some code is rewritten, but the percentage of modified design (MD), the percentage of modified code (MC), the percentage of external code to be integrated (EI), amount of software understanding required (SU, computed taking into account the degree of unfamiliarity with software), the assessment and assimilation effort (AA) and the number of source lines of code to be adapted (ASLOC) are imprecisely known, and given by neutrosophic numbers, then applying the post-architecture model [43], it follows:

$$\mathrm{ESLOC} = \mathrm{ASLOC}\ (\mathrm{AA} + \mathrm{SU} + 0.4\mathrm{MD} + 0.3\mathrm{MC} + 0.3\mathrm{CI})/100.$$

Hence, the equivalent number of lines of new code (ESLOC) is obtained as a neutrosophic number. After deneutrofication, the crisp value can be obtained by *min* value or *quartile*—based scheme.

If ASLOC = 3200, AA = 2 + 2I, SU = 15 + 3I, DM = 15 + 5I, CM = 20 + 10I, CI = 50 + 20I, then ESLOC = 1408 + 512I.

In a similar way, other COCOMO equations [44] can be considered and used to derive various economical and quality indicators.

If the function points are used along the entire life cycle of software [13], and the number of function points between released versions are imprecisely known and modeled by neutrosophic numbers, the power of neutrosophic numbers is required.

Let be z = a + bI one neutrosophic number. To compute $z^{1.25}$, the following method can be used.

Since 1.25 = 5/4 it follows that $(a + bI)^{5/4} = (u + vI)$ and $(a + bI)^5 = (u + vI)^4$ for some u and v to be obtained.

Hence $a^5 + (5a^4b + 10a^{32}_{b} + 10a^2b^3 + 5ab^4 + b^5)I = u^4 + (4u^3v + 6u^2v^2 + 4uv^3 + v^4)I$, with $u = a^{5/4}$, and $(a + b)^5 - a^5 = (u + v)^4 - u^4$. Therefore $(a + b)^5 = (u + v)^4$, with $v = (a + b)^{5/4} - a^{5/4}$.

In a similar way, we obtain $(a + bI)^{0.4} = a^{0.4} + ((a + b)^{0.4} - a^{0.4})\ I$, a formula useful to derive the approximate development schedule in calendar months.

For a software having 2000 + 3000I function points, using the rules of Jones [13], it follows a development team of size 13.33 + 20I (about 23.33 full time personnel, using the median deneutrofication), while the size of maintenance team is 1.33 + 2I (about 2.33 persons). The approximate development schedule in calendar months is 90.21 + 9.26I, about 25.54 months by median deneutrofication. Also, the software defect potential is given by $(2000 + 3000\text{I})^{1.125} = 5172 + 9327\text{I}$, with a median deneutroficated value of about 9835.55.

Other rules formulated by Jones in [13], can be used in neutrosophic context.

6 Conclusions

This paper investigates the usage of classical and recent paradigms of computational intelligence to address some topics in software engineering: software requirements engineering, software design and software testing, software reliability allocation and optimization. Finally, neutrosophic computational schemes are proposed to evaluate various economic and quality indicators based on COCOMO model and Jones rules.

The proposed approach can be used with *min* deneutrofication operator and the results will be identical with those obtained by the classical approach (as an optimistic and theoretical view). However, using a *quartile* deneutrofication, the results can be closed to the values in practical software management.

References

1. Aguilar-Ruiz JS, Ramos I, Riquelme JC, Toro M (2001) An evolutionary approach to estimating software development projects. Inf Softw Technol 43(14):875–882
2. Arcuri A, Yao X (2014) Coevolutionary automatic programming for software development. Inf Sci 259:412–432
3. Atanassov KT (2016) Review and new results on intuitionistic fuzzy sets. Mathematical foundations of artificial intelligence seminar, Sofia, 1988, Preprint IM-MFAIS-1–88. Reprinted: Int J Bioautomation 20(S1):S7–S16
4. Bastani F, Pasquini A (1994) Assessment of a sampling method for measuring safety-critical software reliability. In: Proceedings of the 5th international symposium on software reliability engineering, 6–9 Nov, Monterey. IEEE Computer Society Press
5. Berndt D, Fisher J, Johnson L, Pinglikar J, Watkins A (2003) Breeding software test cases with genetic algorithms. In: Proceedings of the 36th annual Hawaii international conference on system sciences. https://doi.org/10.1109/HICSS.2003.1174917
6. Bisi M, Goyal NK (2015) Early prediction of software fault-prone module using artificial neural network. Int J Perform Eng 11(1):43–52
7. Dawson CW (1998) An artificial neural network approach to software testing effort estimation. Trans Inf Commun Technol 20. WIT Press
8. Feldt R, de Oliveira Neto FG, Torkar R (2018) Ways of applying artificial intelligence in software engineering. In: Proceedings of RAISE'18, Gothenburg, Sweden, ACM. https://doi.org/10.1145/3194104.3194109
9. Fenton NE, Pfleeger SL (1996) Software metrics: a rigorous and practical approach. Thomson
10. Fernandez DM (2018) Supporting requirements-engineering research that industry needs: the NaPiRE initiative. IEEE Softw 35(1):112–116
11. Howden WE, Huang Y (1994) Software trustability. In: Proceedings of the fifth International symposium on soft-ware reliability engineering, Monterey, California, pp 143–151. IEEE Computer Society Press

12. Jones TC (2017) The mess of software metrics. http://www.namcook.com/articles.html. Version 9.0
13. Jones TC (1998) Estimating software costs. McGraw-Hill
14. Kamsties E (2005) Understanding ambiguity in requirements engineering. In: Aurum A, Wohlin C (eds) Engineering and managing software requirements. Springer, Berlin, Heidelberg
15. Klees G, Ruef A, Cooper B, Wei S, Hicks M (2018) Evaluating fuzz testing. In: Proceedings of the ACM conference on computer and communications security, pp 2123–2138
16. Kruse R, Borgelt C, Braune C, Mostaghim S, Steinbrecher M (2016) Computational intelligence. A methodological introduction, 2nd edn. Springer
17. Madsen H, Albeanu G, Popenţiu Vlădicescu F (2012) An intuitionistic fuzzy methodology for component-based software reliability optimization. Int J Perform Eng 8(1):67–76
18. Madsen H, Thyregod P, Burtschy B, Albeanu G, Popentiu-Vladicescu F (2007) On using chained neural networks for software reliability prediction. In: Proceedings of the European safety and reliability conference 2007, ESREL 2007—risk, reliability and societal safety, vol 1, pp 411–417
19. Madsen H, Thyregod P, Burtschy B, Popentiu-Vladicescu F, Albeanu G (2006) On using soft computing techniques in software reliability engineering. Int J Reliab Qual Saf Eng 13(1):61–72
20. Madsen H, Thyregod P, Burtschy B, Albeanu G, Popentiu F (2006) A fuzzy logic approach to software testing and debugging. In: Guedes Soares C, Zio E (eds) Safety and reliability for managing risk, pp 1435–1442
21. McGraw G, Michael C, Schatz M (2001) Generating software test data by evolution. IEEE Trans Software Eng 27(12):1085–1110
22. Meziane F, Vadera S (2010) Artificial intelligence in software engineering—current developments and future prospects. In: Meziane F, Vadera S (eds) Artificial intelligence applications for improved software engineering development: new prospects, information science reference, pp 278–299. https://doi.org/10.4018/978-1-60566-758-4.ch014
23. Michael CC, McGraw GE, Schatz MA, Walton CC (1997) Genetic algorithms for dynamic test data generation. In: Proceedings 12th IEEE international conference automated software engineering. https://doi.org/10.1109/ASE.1997.632858
24. Osman MH, Zaharin MF (2018) Ambiguous software requirement specification detection: an automated approach, RET2018, June 2018, Gothenburg, Sweden http://ret.cs.lth.se/18/downloads/RET_2018_paper_9.pdf
25. Patton RM, Wu AS, Walton GH (2003) A genetic algorithm approach to focused software usage testing. In: Khoshgoftaar TM (ed) Software engineering with computational intelligence, vol 731. The Springer international series in engineering and computer science. Springer, Boston, MA, pp 259–286
26. Pedrycz W (2002) Computational intelligence as an emerging paradigm of software engineering. In: ACM proceedings of SEKE'02, pp 7–14
27. Petrică L, Vasilescu L, Ion A, Radu O (2015) IxFIZZ: integrated functional and fuzz testing framework based on Sulley and SPIN. Rom J Inf Sci Technol 18(1):54–68
28. Popentiu-Vladicescu F, Albeanu G (2017) Recent advances in artificial immune systems: models, algorithms, and applications. In: Patnaik S (ed) Recent developments in intelligent nature-inspired computing. IGI Global, Hershey, PA, pp 92–114. https://doi.org/10.4018/978-1-5225-2322-2.ch004
29. Popentiu-Vladicescu F, Albeanu G, Madsen H (2019) Improving software quality by new computational intelligence approaches. In: Pham H (ed) Proceedings of 25th ISSAT international conference "Reliability & Quality in Design", pp 152–156
30. Rech J, Althoff K-D (2004) Artificial intelligence and software engineering: status and future trends. Spec Issue Artif Intell Softw Eng 3:5–11
31. Smarandache F (2007) A unifying field in logics: neutrosophic logic. Neutrosophy, neutrosophic set, neutrosophic probability and statistics, 6th edn. ProQuest information & learning, Ann Arbor (2007)
32. Smarandache F (2016) Subtraction and Division of neutrosophic numbers. Crit Rev XIII:103–110

33. Sorte BW, Joshi PP, Jagtap V (2015) Use of artificial intelligence in software development life cycle—a state of the art review. IJAEGT 3(3):398–403
34. Sultanov H, Hayes JH (2013) Application of reinforcement learning to requirements engineering: requirements tracing. RE 2013, Rio de Janeiro, Brazil, Research Track. http://selab.netlab.uky.edu/homepage/publications/RE-hakim.pdf
35. Venkataiah V, Ramakanta M, Nagaratna M (2017) Review on intelligent and soft computing techniques to predict software cost estimation. IJAER 12(22):12665–12681
36. Wang H, Smarandache F, Zhang Y-Q, Sunderraman R (2005) Interval neutrosophic sets and logic: theory and applications in computing, Hexis
37. Wappler S, Lammermann F (2005) Using evolutionary algorithms for the unit testing of object-oriented software. In: GECCO '05 proceedings of the 7th annual conference on genetic and evolutionary computation. ACM https://doi.org/10.1145/1068009.1068187
38. Wójcicki B, Dabrowski R (2018) Applying machine learning to software fault prediction. e-Inform Softw Eng J 12(1):199–216. https://doi.org/10.5277/e-inf180108
39. Xing B, Gao W-J (2014) Innovative computational intelligence: a rough guide to 134 clever algorithms. Springer
40. Zadeh LA (1965) Fuzzy sets. Inf Control 8:338–353
41. Zadeh LA (1994) Fuzzy logic, neural networks, and soft computing. Commun ACM 37(3):77–84
42. Computational intelligence society page. https://cis.ieee.org/
43. How-harmful-can-be-ambiguous-software-requirements. Cigniti's requirement testing framework (RTF). https://www.cigniti.com/
44. COCOMO manual. https://sunset.usc.edu/research/COCOMOII/Docs/modelman.pdf

Grigore Albeanu received his Ph.D. in Mathematics in 1996 from University of Bucharest, Romania. During the interval 2004–2007 he was the head of the UNESCO IT Chair at University of Oradea, and participated as principal investigator in a NATO project. Grigore Albeanu has authored or co-authored over 120 papers and 10 educational textbooks in applied mathematics and computer science. Since 2007, he is professor of computer science at "Spiru Haret" University in Bucharest. His current research interests include different aspects of scientific computing, including soft computing, modelling and simulation, software reliability, virtual reality techniques and E-Learning.

Henrik Madsen received a Ph.D. in Statistics at the Technical University of Denmark in 1986. He was appointed Ass. Prof. in Statistics (1986), Assoc. Prof. (1989), and Professor in Mathematical Statistics with a special focus on Stochastic Dynamical Systems in 1999. In 2017 he was appointed Professor II at NTNU in Trondheim. His main research interest is related to analysis and modelling of stochastic dynamics systems. This includes signal processing, time series analysis, identification, estimation, grey-box modelling, prediction, optimization and control. The applications are mostly related to Energy Systems, Informatics, Environmental Systems, Bioinformatics, Biostatistics, Process Modelling and Finance. He has got several awards. Lately, in June 2016, he has been appointed Knight of the Order of Dannebrog by Her Majesty the Queen of Denmark, and he was appointed Doctor HC at Lund University in June 2017. He has authored or co-authored approximately 500 papers and 12 books. The most recent books are Time Series Analysis (2008); General and Generalized Linear Models (2011); Integrating Renewables in Electricity Markets (2013), and Statistics for Finance (2015).

Florin Popenţiu-Vlădicescu was born on 17 September 1950, graduated in Electronics and Telecommunications from University POLITEHNICA of Bucharest in 1974, holds a Ph.D. in Reliability since 1981. Also, he is Associated Professor to University "Politehnica" of Bucharest, Faculty of Automatic Control and Computer Science. He is the founder of the first "UNESCO

Chair of Information Engineering", in UK, established at City University London, in 1998. He published over 150 papers in international journals and conference proceedings. Also he is author of one book, and co-author of 4 books. He has worked for many years on problems associated with software reliability and has been Co-Director of two NATO Research Projects, in collaboration with Telecom ParisTech and Danish Technical University (DTU). He is an independent expert to the European Commission—H2020 programme, for Net Services—Software and Services, Cloud. He is currently Visiting Professor at renowned European and South Asian universities. He was elected Fellow of the Academy of Romanian Scientists in 2008.