Article

Fuzzy Logic Testing Approach for Measuring Software Completeness

Nayyar Iqbal and Jun Sang *

School of Big Data & Software Engineering, Chongqing University, Chongqing 401331, China; nayyariqbal@cqu.edu.cn
* Correspondence: jsang@cqu.edu.cn; Tel.: +86-139-8369-7592

Abstract: Due to advancements in science and technology, software is constantly evolving. To adapt to newly demanded requirements in a piece of software, software components are modified or developed. Measuring software completeness has been a challenging task for software companies. The uncertain and imprecise intrinsic relationships within software components have been unaddressed by researchers during the validation process. In this study, we introduced a new fuzzy logic testing approach for measuring the completeness of software. We measured the fuzzy membership value for each software component by a fuzzy logic testing approach called the fuzzy test. For each software component, the system response was tested by identifying which software components in the system required changes. Based on the measured fuzzy membership values for each software component, software completeness was calculated. The introduced approach scales the software completeness between zero and one. A software component with a complete membership value indicates that the software component does not require any modification. A non-membership value specifies that the existing software component is no longer required in the system or that a new software component is required to replace it. The partial membership value specifies that the software component requires few new functionalities according to the new software requirements. Software with a partial membership value requires partial restructuring and design recovery of its components. Symmetric design of software components reduces the complexity in the restructuring of software during modification. In the study, we showed that by using the introduced approach, high-quality software that is faultless, reliable, easily maintained, efficient, and cost-effective can be developed.

Keywords: functionality; fuzzy logic; fuzzy test; modification; software completeness; software components; system response

1. Introduction

Soft computing techniques such as fuzzy logic have been extensively used in software engineering to address uncertainty and imprecision [1–4]. In this study, we measured software completeness by introducing the fuzzy logic testing approach. In this study, we classified the software requirements into different types. The fuzzy membership value scaled the software completeness between zero and one. The software completeness value exactly extracted the software components that required changes according to the latest requirements. By this approach, complete membership, partial membership, and non-membership values for the software components were determined by validating the software components against its requirements.

A software component, as defined by Pang [5], is “a software unit of functionality that manages a single abstraction.” In the literature, completeness has been considered as an essential quality attribute. Davis et. al., in [6], measured the quality of software require-
ments by quality attribute (e.g., completeness) and, in [7], forward engineering completeness for software was measured, whereas incompleteness can cause unsuccessful effects [8,9]. Software completeness is defined as the state of software having all the required software components with appropriate functionality and features and it certifies that no functionality or feature is left out.

The evolution of software occurs due to technology drivers and business needs. During the software evolution process, its functionalities and features continuously change, and as software size increases, ultimately, its complexity also increases. Software products are changed based on global changes in business. The change process comprises modifications, additions, and deletions of software components or modules in the software system. Software generally grows in terms of features, so it must be designed to accommodate the demanded changes and satisfy a set of new requirements [7,10–12].

A feature is defined as [13] “an optional or incremental unit of functionality.” \( S_F = \{f_1, f_2, f_3 \ldots f_{r-1}, f_r\} \), where \( S_F \) represents a set of features within the system. In this study, each feature is designed based on a requirement or set of requirements \( R \) and is verified with the expected system response \( SR_{kC_i} \) to a specific status, \( f_q = (R, SR_{kC_i}) \), where (1 \( \leq k \leq n \)) and (1 \( \leq q \leq m \)). \( SR_{kC_i} \) denotes the system response tested for the \( i^{th} \) software component \( C_p \) that contains \( n \) number of system responses. System response is defined as the set of all possible expected output results for each input data, to determine whether a feature is correctly working or not [14–18].

Consider an example where a client asks the software engineer about an object F: what is F made of \( \rightarrow \) component, what can F do \( \rightarrow \) feature. Examples of components include fee checkers, registered students, and publications. Examples of features include the following: can detect the total amount of fees paid by the students, can determine the total number of students registered in a specific course during a given semester at a university, can generate reports about faculty publications.

According to GenVoca theory, now called feature-oriented software development, features are implemented as applied functions, whereas the compositions of features are implemented as composite functions, which are signified by •. In most cases, the order of composition does not affect the process, but in other cases, the composition order matters [19].

Software requirements depict functionalities and features of the system to be developed in descriptive form, as well as constraints on the services of the system. The software requirement known as completeness is defined as a set of requirements or a single requirement that satisfies the demanded quality, such that no information regarding the software to be developed is left out [9,20]. Fuzzy logic is based on membership values, where non-membership is represented by zero, partial membership values vary between zero and one, and complete membership is represented by one [21–23].

The novel fuzzy logic testing approach identified the missing functionalities, features, and new software requirements. The software was developed by component-based development. Therefore, by applying the fuzzy test, errors were extracted from the exact functionality. The developed software components were reliable, upgradable, cost effective, reusable, and time effective and were developed with improved quality. In this study, the authors classified the software requirements into functional, non-functional, inverse, non-inverse, content, and non-content requirements. The classification of software requirements helped to represent the software requirements in the precise form and in identification of imprecise software requirements.

The rest of the sections in this paper are organized as follows: Section 2 defines the preliminary concepts, Section 3 describes related work, Section 4 describes fuzzy membership value of the software components, Section 5 explains the measurement approach.
for software completeness, Section 6 describes materials and methods, Section 7 shows results, and finally, Section 8 concludes the paper.

2. Preliminaries

In this section, we define the necessary concepts related to software requirements, fuzzy functionality similarity between software modules, completeness of software components, linguistic variables, proof of software security level, and programming language faults.

2.1. Software Requirements

Extracting rules or requirements from natural language has been a challenging task. In this study, we used the modality concept to define the requirement types. Modality, as defined by Sleimi et al., is [24] “a verb indicating the modality of the action, e.g., must, shall.”

i. Functional Requirements and Non-Functional Requirements: Functional requirements (FR) describe how the system should act according to statements of services defined in the software specification. One example of a FR is the following: the system shall allow users to search all sets of books related to course work or select subsets from it. In this modality, shall → functional requirement.

Non-Functional Requirements (NFR) are limitations on system functions or services such as timing, security, usability, availability, performance, etc. NFR are applied to the entire system, not to specific features or services of the system. One example of an NFR is the following: the system shall be available 24/7. Modality: the requirement in which “shall” is used to represent the limitations on system functions → non-functional requirement.

Data privacy is a non-functional requirement that assures the personal data are secure from unauthorized access. In data privacy, the following types of information are considered sensitive, by legal mandates and also by the general public [25]: personally identifiable information (PII), personal health information (PHI), personally identifiable financial information (PIFI), and student records. Consider an example of an NFR related to data privacy: the system shall protect personal data from unauthorized users. Timing issues: This is an outset with the inclusion of timing definition in the software requirements. In testing timing issues, the task of the software tester is to find the temporal error in the input situations, if the execution takes the shortest or longest time to complete the tasks [26].

ii. Inverse Requirements and Non-Inverse requirements: As defined in [27,28], these can be functional requirements or non-functional requirements in which customers specify “what the system shall not do” or the system shall display a text box in red if the user does not enter the data according to the specified condition of the system. Inverse requirements that describe the functional aspects of the system are represented by IR, whereas those that describes non-functional aspects of the system are known as non-inverse requirements, represented by NIR. An example of an IR is the following: the system shall not allow users to enroll in more than 12 credit hours in a semester. An example of an NIR is the following: the system shall not be available from 12:00 a.m. to 3:00 a.m. The modality “shall” in negative form (shall not) → inverse requirements, whereas the requirement in which it is used to represent the limitations on system functions → non-inverse requirements.

iii. Content Requirements and Non-Content Requirements: As defined by the authors, these can be functional requirements or non-functional requirements in which customers specify “what the system must do.” Content requirements are used for customized software. Content requirements are represented by CR, whereas non-content requirements are represented by NCR. The purpose of content requirements is to elaborate on the differences between the software requirements demanded by different organizations. A CR would be as follows: the system must ask the student to complete the teacher evaluation form before redirecting him/her to the main screen. An NCR would
be as follows: the system must respond to user requests within 3 s. The statements of requirement with the modal verb “must” are specified as content requirements, whereas the requirement in which the modality “must” is used to represent the limitation on the system, e.g., performance (within 3 s) → non-content requirement.

In this study, each software component was validated against its software requirements. The software components were developed by using the process of component-based development. To determine whether the software components were required in the software, we used a requirements validation framework [7]. As symmetry creates balance in design, therefore, each software component was developed to be symmetric in design, whereas a few software components were developed to be asymmetric in order to grab attention. Software completeness denotes the membership value that covers the requirements in new software. For this, we used a fuzzy test. Software requirements were gathered from the users of the software in natural language. The requirements were classified using the concept of modality. The inverse and content requirements were classified based on different demanded features and functionalities or constraints within the same software component or overall software.

2.2. Fuzzy Functionality Similarity Between Software Modules

Consider software S, which consists of n software components Ci. Rj represents the ith software component, whereas j represents the software requirement that corresponds to a particular functionality and feature of a software component. FCk represents the functionality of the component, i represents the software component and k represents the functionality of the ith software component. In this study, we tested two software modules, Mf and Mp. We used the fuzzy if–then rule to determine the functionality similarity between software modules.

\[
\text{If } R_j^i (M_f) \text{ is } C_m \text{ then } R_j^i (M_p) \text{ is } C_m
\]

Figure 1 represents fuzzy functionality similarity, and an increase in color shade denotes an increase in the functionalities within software components, whereas white represents software components with single functionality.

![Figure 1. Fuzzy functionality similarity.](image)

2.3. Completeness of Software Components

\[
(R_1^1, R_2^1, R_3^1, \ldots, R_n^1) \in S
\]

\[
R_j^i \Leftrightarrow FC_k
\]
As in the experimental requirements, \( R_1, R_2, R_3, R_4, R_5, R_6, \ldots, R_n \) represent \( FR_1 \) & \( NFR_1 \), \( FR_2 \) & \( NFR_2 \), \( CR_3 \) & \( NCR_3 \), \( IR_4 \) & \( NIR_4 \), \( IR_5 \) & \( NIR_5 \), \( FR_6 \) & \( NFR_6 \), \( FR_7 \) & \( NFR_7 \), \( FR_8 \) & \( NFR_8 \), \( CR_9 \) & \( NCR_9 \), \( FR_{10} \) & \( NFR_{10} \), respectively.

\[
\begin{align*}
FR_1 & \neq NFR_1 \\
FR_2 & \neq NFR_2 \\
CR_3 & \neq NCR_3 \\
IR_4 & \neq NIR_4 \\
& \vdots \\
FR_n & \neq NFR_n
\end{align*}
\]

Each software component consists of one or more features and functionalities within a single component.

\[
\sum_{i=1}^{n} FC_i = FC_1 + FC_2 + FC_3 + \ldots + FC_n
\]

2.3.1. Complete Membership

Considering the software component \( C_v \) with three functionalities \( FC_1^v, FC_2^v \) and \( FC_3^v \), the expected system responses are represented by \( SR_1C_v, SR_2C_v \) and \( SR_3C_v \), respectively. All functionalities of the software component are represented in blue if the software component possesses a complete membership value. Figure 2 represents a software component that possesses complete membership.

![Figure 2. Complete membership.](image)

2.3.2. Partial Membership

Considering the software component \( C_y \) with three functionalities \( FC_1^y, FC_2^y \) and \( FC_3^y \), the expected system responses are represented by \( SR_1C_y, SR_2C_y \) and \( SR_3C_y \), respectively. This software component possesses a partial membership value; the functionalities with complete membership values are represented in blue, whereas the functionality with a partial membership value is represented in brown. Figure 3 represents a software component that possesses partial membership.

![Figure 3. Partial membership.](image)

2.3.3. Non-Membership

Considering the software component \( C_z \) with three functionalities \( FC_1^z, FC_2^z \) and \( FC_3^z \), the expected system responses are represented by \( SR_1C_z, SR_2C_z \) and \( SR_3C_z \), respectively. This software component possesses a non-membership value, and functionalities with non-membership values are represented in green. Figure 4 represents a software component that possesses non-membership.

![Figure 4. Non-membership.](image)
2.4. Linguistic Variable

Linguistic variable [29] values are in sentences or words of natural language. For example, “change” is linguistic variable, whereas “minor changes”, “major changes” and “complex changes” are its values. It represents the semantic concepts in mathematical representation [30], which consist of more than one term, called a fuzzy set. Linguistic variable uses the values of fuzzy variable. Consider the linguistic variable: (c, T(c), U, M), the variable name is represented by c, whereas its term set is denoted by T(c), and each value defined in U is a fuzzy variable. M: each variable consists of associated semantic rules (membership), c: “change” is a linguistic variable.

2.5. Proof of Software Security Level

Software security levels are tested in different perspectives during software development in order to build secure software that is invulnerable before it deployment. As described by Murray [31], a software program is insecure and imperfect if it “deviates from its intended behavior (e.g., arbitrary code execution), or reveal sensitive data (data leakage), or modify data that should remain protected (integrity violation).” Researchers have conducted research on proving secure software, such as identifying security flaws [32] in protocols and programs and also on transport layer security (TLS) implementations [33], operating system kernels [34], cryptographic algorithms [35], etc.

In order to make the software secure, we tested the software components with the fuzzy test for all possible system responses ($SR_iC_j$) by entering different input data, e.g., actions ($A_1$–$A_n$). The input data are entered implicitly or explicitly in order to check all the possible positive and negative system responses.

2.6. Programming Language Faults

Software program $P_3$ consists of $n$ number of software components $C_i$, (1 ≤ $i$ ≤ n), in which software component $C_{34}$ in $P_3$ is faulty, if $P_3$ cannot be correctly verified with respect to existing implementation of $C_{34}$, but it can be verified by the implementation specification for $C_i$. Figure 5 represents types of programming language faults.

![Diagram of Types of Programming Language Faults](image-url)
Types of programming language faults are as follows [17].

2.6.1. Algorithm Fault

This fault occurs in the program when the logic of the algorithm does not deliver the correct output (result) for a certain input due to incorrect processing, e.g., the system should display a list of journals recognized by Web of Science if the user selects the Web of Science indexing option but the system displays a list of journals recognized by EI (Engineering Index) due to a fault.

2.6.2. Computational Fault

This occurs when the implementation of program is wrong or the program is not capable of computing the desired output. The system displays unexpected output due to combining the floating point and integer variables.

2.6.3. Syntax Fault

This fault occurs in the program due to incorrect syntax,

\[
\text{e.g., } \quad \gg y = y \times ((3 + 5 \times y); \quad \text{fault: } \% \text{ extra parenthesis}
\]

\[
\gg x = \text{`Chongqing} \quad \text{fault: } \% \text{ missing quote}
\]

2.6.4. Documentation Fault

The program documentation describes the software in the form of illustrations and written text. It also describes how to operate it or use it. A documentation fault occurs when this does not match with the program.

2.6.5. Overload Fault

During the program implementation, a stack, array or queue is used for memory purposes. If more elements are added than its limited size, then an overload fault occurs in the program.

2.6.6. Timing Fault

If the system does not respond to a user due to the occurrence of a failure in the program, then the system reports a timing fault.

2.6.7. Hardware Fault

Software companies are trendsetters for the hardware industry. Hardware is updated in order to support new features and functionalities of the software. This fault occurs if the hardware does not work properly with the software. This problem can be avoided if hardware specifications are completely specified in the technical feasibility.

2.6.8. Software Fault

Software faults occur if the software does not function properly, or is not supported by the operating system, such Microsoft Windows, Linux, Apple macOS, Apple iOS or Android.

2.6.9. Omission Fault

This fault occurs “when we fail to enter correct information” [17], e.g., in a program in which a variable is not initialized (fault: int y =; correct: int y = 10;).

2.6.10. Commission Fault

This fault occurs “when we enter something into a representation that is incorrect” [17], e.g., a program in which char is initialized to int variable incorrectly (int n = a;).
3. Related Work

In [36], MacDonell and Gray applied a fuzzy logic modeling tool set to estimate the size of a product and to determine the risks of the project. Kacprzvk and Yager [30] used fuzzy logic on data sets for the extraction of linguistic summaries, and the researchers also focused on which methods could be applied to assess the goodness of the linguistic summary. According to Barry Boehm [37], software contains fixed features such as the database size, product complexity and required reliability that are not controllable by management.

Most size estimations are calculated based on imperfect, incomplete knowledge and assumptions about software features [38]. In software engineering practices, it has been observed that more user engagement in the software development process increases the software success rate [39]. Researchers [40–42] concluded that user feedback plays a vital role in software quality and in detecting new design concepts for software evolution.

In the literature, researchers concluded that fuzzy logic can be used in different phases of the system development life cycle (SDLC), such as for system design [43], software testing [44,45], estimation of product size, estimation of product risk related to which diverse input parameters are used [46], etc. In [47], researchers applied fuzzy logic for the prioritization of test cases which minimizes the cost of regression testing. Figure 6 represents the extraction of different solutions/activities when fuzzy logic is applied in different phases of the system development life cycle (SDLC) [48–56].

![Figure 6. Software engineering and fuzzy logic.](image)

In the literature, different techniques for gathering requirements have been introduced by researchers [57–60], such as one-on-one interviews, group interviews, facilitated sessions, joint application design (JAD), a group support system (GSS), prototyping, questionnaires, use cases, following people around, requests for proposals (RFPs) and brainstorming. In a survey conducted with software engineers, these techniques were also found to possess limitations. Therefore, in this study, we used a requirements validation framework [7] for the validation of software components. In [61], researchers introduced tsDetect which is an open source tool used to detect test smells. tsDetect can detect fol-
lowing types of test smells: assertion roulette, conditional test logic, constructor initialization, default test, duplicate assert, eager test, empty test, exception handling, general fixture, ignored test, lazy test, magic number test, mystery guest, redundant print, redundant assertion, resource optimism, sensitive equality, sleepy test and unknown test. In this study, we did not find any similar approaches in which the completeness of software was measured by our introduced approach.

4. Fuzzy Membership Value of the Software Components

To measure the fuzzy membership value for each software component, we used the introduced fuzzy logic testing approach called the fuzzy test.

Definition

Let $m_{k,i}$ represent the membership value of the system response for software component $C_i$, which is the ratio of successful system responses $k$ in $C_i$ to the total number of expected system responses during the occurrence of $C_i$. The fuzzy membership value $m_i$ is the sum of $m_{k,i}$ for the software component $C_i$ that contains $k$ system responses.

$$m_i = \sum_{k=1}^{n} m_{k,i} = \sum_{k=1}^{n} \frac{SR_k}{SRC_i} \tag{1}$$

Rules for Fuzzy Membership Value

if ($m_{k,i} = SRC_i$) then
  $m_i = 1$
else if ($m_{k,i} \neq SRC_i$) then
  $0 < m_i < 1$
else
  $m_i = 0$
end if

5. Software Completeness

In this study, we measured the software completeness with our introduced quantitative approach. Each software component was validated against its corresponding requirements. By this approach, missing functionalities or features and new software requirements were identified. $S_c$, $S_f$, $S_E$, $C_C$, $C_P$, $C_N$ and $C_T$ represent the software completeness, software components completeness index, engineering of software (reverse or forward engineering), complete membership of the software components, partial membership of the software components, non-membership of the software components and the total number of software components, respectively.

$$S_i = [1 - [(C_C/C_T) + (C_P/C_T) + (C_N/C_T)]] \tag{2}$$

$$S_c = S_E + S_f \tag{3}$$

By substituting Equation (2) into Equation (3),

$$S_c = S_E + [1 - [(C_C/C_T) + (C_P/C_T) + (C_N/C_T)]]$$

5.1. Proof

In this section, the authors prove the software completeness equation. According to Lotif Zadeh [23], fuzzy logic is defined as:

$$\text{membership value} \rightarrow [0, 0.01, 0.02, \ldots, 0.98, 0.99, 1]$$

In this, zero represents non-membership, one represents complete membership and the value between zero and one represents partial membership. Fuzzy logic evolved in the context of the theory of fuzzy sets. Fuzzy sets are extension of Boolean and crisp sets.
Fuzzy membership values for each software component are calculated by a fuzzy logic testing approach (fuzzy test). If the fuzzy membership value is equal to one, then it is assigned to $C_c$, else if the fuzzy membership value is equal to partial (varies between zero and one), then it is assigned to $C_p$, else it is assigned to $C_n$.

Therefore,

$$0 \leq m \leq 1$$

Therefore, as software membership value is determined by the $S_c$ equation, by substituting the values of $C_c$, $C_p$, $C_n$ and $S_E$.

$$0 \leq S_E + \left[1 - \left(\frac{C_c}{C_T} + \frac{C_p}{C_T} + \frac{C_n}{C_T}\right)\right] \leq 1.$$

5.1.1. Complete Membership Value

If the software does not require any changes due to evolution of the business or the operating environment, then the complete membership value of the software components $C_c$ is equal to the membership value of the total software components $C_T$. The partial membership value of a software component $C_p$ becomes equal to the non-membership value of the software components because the software does not require any minor or major modifications during the validation process. If the software possesses complete membership, then $C_c = C_T$, $C_p = 0$ and $C_n = 0$, whereas $C_p = C_n = 0$. $C_c = C_T$ because the software is not missing any features nor does it require any newly demanded features. In this section, we substitute the values of $C_c$, $C_p$ and $C_n$ in Equation (2), to prove that the software membership value is equal to the non-membership value.

$$S_c = \left[1 - \left(\frac{C_c}{C_T} + \frac{C_p}{C_T} + \frac{C_n}{C_T}\right)\right] = \left[1 - \left(0 + 0 + 0\right)\right] = 1 - 0 = 0$$

Hence, this proves that the membership value is zero when the software exhibits completeness equal to one. In this case, $S_E = 0$ because the software does not require re-structuring.

5.1.2. Non-membership Value

If all the software components do not support the functionalities or features demanded in the new requirements due to low business values and low component quality, then $C_n = 0$ for all software components. If the software possesses a non-membership value, then $C_n = 0$, $C_p = 0$ and $C_c = 0$. The new software is completely developed by using a forward engineering approach. In this case, the components with non-membership value $C_n$ are completely satisfied/validated by the user of the software. By substituting the values of $C_c$, $C_p$ and $C_n$ in Equation (2),

$$S_n = \left[1 - \left(\frac{C_c}{C_T} + \frac{C_p}{C_T} + \frac{C_n}{C_T}\right)\right] = \left[1 - \left(0 + 0 + 0\right)\right] = 1 - 0 = 1$$

Hence, this proves that the membership value is one when the software exhibits a membership value equal to zero. $S_E = 1$, when the existing software does not support the new user requirements, the software is outdated or the software does not support changes, and the system is a candidate for scrapping because keeping it in operation would be expensive [6,27,53].

5.1.3. Partial Membership Value

If the software requires minor or major changes in its components according to new user requirements, then the membership value of the software is equal to partial membership. The partial membership value is determined by validating each software component against its corresponding software requirements. Therefore, software membership is a combination of complete membership, partial membership and non-membership values,
whereas $S_c$ determines the partial membership value of software. In this case, the software is updated by a reverse engineering approach. Figure 7 represents the software components in the conceptual model.

![Figure 7. Conceptual model of software components.](image)

According to this figure, the $C_C$, $C_P$, $C_N$ and $C_T$ values are calculated as

\[ C_i = C_1 + C_2 + C_3 + \ldots + C_{24} \]

\[ C_1 = FC_1 = 1 \rightarrow \text{complete membership}; \]
\[ C_2 = (FC_1^2 + FC_2^2) = ((0.3 + 1)/2) = 0.65 \rightarrow \text{partial membership}; \]
\[ C_3 = FC_3 = 0.9 \rightarrow \text{partial membership}; \]
\[ C_4 = (FC_4^4 + FC_2^6) = ((1 + 0)/2) = 1 \rightarrow \text{complete membership}; \]
\[ C_5 = (FC_5^6 + FC_2^8) = ((0.1 + 0.6)/2) = 0.35 \rightarrow \text{partial membership}; \]
\[ C_6 = FC_6 = 0 \rightarrow \text{no membership}; \]
\[ C_7 = (FC_7^7 + FC_2^9) = ((0 + 0 + 0)/3) = 0 \rightarrow \text{no membership}; \]
\[ C_8 = (FC_8^8 + FC_2^{10} + FC_2^{11} + FC_2^{12}) = ((1 + 1)/2) = 1 \rightarrow \text{complete membership}; \]
\[ C_9 = (FC_9^{11} + FC_2^{13}) = ((0.7 + 0.6)/2) = 0.65 \rightarrow \text{partial membership}; \]
\[ C_{10} = FC_1 = 1 \rightarrow \text{complete membership}; \]
\[ C_{11} = FC_1^2 = 0.6 \rightarrow \text{partial membership}; \]
\[ C_{12} = FC_1^3 = 1 \rightarrow \text{complete membership}; \]
\[ C_{13} = FC_1^4 = 0.9 \rightarrow \text{partial membership}; \]
\[ C_{14} = FC_1^5 = 1 \rightarrow \text{complete membership}; \]
\[ C_{15} = FC_1^6 = 0.9 \rightarrow \text{partial membership}; \]
\[ C_{16} = FC_1^7 = 1 \rightarrow \text{complete membership}; \]
\[ C_{17} = FC_1^8 = 0.9 \rightarrow \text{partial membership}; \]
\[ C_{18} = FC_1^9 = 1 \rightarrow \text{complete membership}; \]
\[ C_{19} = FC_1^{10} = 0.9 \rightarrow \text{partial membership}; \]
\[ C_{20} = FC_1^{11} = 1 \rightarrow \text{complete membership}; \]
\[ C_{21} = FC_1^{12} = 0.8 \rightarrow \text{partial membership}; \]
\[ C_{22} = FC_1^{13} = 0.8 \rightarrow \text{partial membership}; \]
\[ C_{23} = FC_1^{14} = 1 \rightarrow \text{complete membership}; \]
\[ C_{24} = FC_1^{15} = 1 \rightarrow \text{complete membership}; \]

By substituting the values in Equation (2),

\[ C_C = C_1 + C_4 + C_{10} + C_{13} + C_{17} + C_{22} = 1 + 1 + 1 + 1 + 1 + 1 = 6 \]
\[ C_P = C_2 + C_3 + C_5 + C_6 + C_9 + C_{11} + C_{14} + C_{15} + C_{16} + C_{18} + C_{19} + C_{20} + C_{21} + C_{23} + C_{24} = 0.65 + 0.9 + 0.93 + 0.35 + 0.85 + 0.65 + 0.9 + 0.6 + 0.8 + 0.5 + 0.45 + 0.35 + 0.85 + 0.8 + 0.33 = 9.91 \]
\[ C_N = C_5 + C_8 + C_{12} = 0 + 0 + 0 = 0 \]
\[
S_i = [1 - [(C_C / C_T) + (C_P / C_T) + (C_N / C_T)]]
\]
\[
S_i = [1 - [(6/24) + (9.91/24) + (0/24)]]
\]
\[
S_i = [1 - (0.25 + 0.41 + 0)] = [1 - 0.66] = 0.34
\]
\[
S_c = S_g + S_i
\]
\[
S_c = 0 + 0.34 = 0.34
\]

In this case, \( S_g = 0 \) because software is updated by the reverse engineering approach. As the partial membership value varies between complete membership and non-membership, \( 0 < \text{partial membership value} < 1 \). Therefore, \( 0 < 0.34 < 1 \), and we obtain partial membership.

Section 6 describes in detail how to calculate the membership value for each software component.

6. Materials and Methods

Measuring software completeness has been a challenging task for software companies. Due to changes in the operating environment, software is changing day by day. In this study, we introduced a fuzzy logic testing approach for measuring software completeness. Software requirements were gathered in natural language and were classified into different types. To measure the fuzzy membership value for each software component, each functionality was tested with the newly demanded system responses.

The dataset in this study consisted of 560 requirements, as shown in Table 1. There were 365 functional requirements, 166 non-functional requirements, 9 inverse requirements, 4 non-inverse requirements, 11 content requirements and 5 non-content requirements. There were thirteen requirement classes of non-functional requirements, named security, usability, portability, availability, capacity, efficiency, reliability, performance, maintainability, accuracy, error-handling, operational and scalability.

| Requirement Type | Requirement Class | Number of Requirements | Percentage |
|------------------|-------------------|------------------------|------------|
| FR               | Functional        | 365                    | 65.18%     |
| NFR              | Security          | 14                     | 2.50%      |
|                  | Usability         | 21                     | 3.75%      |
|                  | Portability       | 10                     | 1.79%      |
|                  | Availability      | 7                      | 1.25%      |
|                  | Capacity          | 11                     | 1.96%      |
|                  | Efficiency        | 12                     | 2.14%      |
|                  | Reliability       | 15                     | 2.68%      |
|                  | Performance       | 13                     | 2.32%      |
|                  | Maintainability   | 15                     | 2.68%      |
|                  | Accuracy          | 8                      | 1.43%      |
|                  | Error-handling    | 10                     | 1.79%      |
|                  | Operational       | 19                     | 3.39%      |
|                  | Scalability       | 11                     | 1.96%      |
| IR               | Inverse           | 9                      | 1.61%      |
| NIR              | Security          | 2                      | 0.36%      |
|                  | Availability      | 1                      | 0.18%      |
|                  | Performance       | 1                      | 0.18%      |
| CR               | Content           | 11                     | 1.96%      |
| NCR              | Security          | 1                      | 0.18%      |
|                  | Usability         | 1                      | 0.18%      |
|                  | Performance       | 1                      | 0.18%      |
|                  | Operational       | 2                      | 0.36%      |
Based on the difference between the previously expected system responses and newly expected system responses, the partial membership value was determined. Other software components were tested based on the system responses. If any new functionality was required within the software component or it was omitted during the development due to any reason, it was considered zero membership $C_N = 0$ during the validation process. In this study, we tested the software components, and Figure 8 represents a fuzzy test for measuring the fuzzy membership value of the $C_S$ software component.

![Fuzzy Test](image)

Figure 8. Fuzzy test for login software component.
The tested login software component consisted of six expected system responses that were successfully tested with system responses $SR_{6C_6}$, $SR_{5C_6}$, $SR_{4C_6}$, $SR_{3C_6}$, $SR_{2C_6}$, and $SR_{1C_6}$. The fuzzy membership value of this software component was $m_6 = 1$.

Figure 9 represents a fuzzy test for measuring the fuzzy membership value of the $C_{19}$ software component.

| Fuzzy Test: $R_{15}^{19}$ |
|---------------------------|
| Test Case Number: 1003    |
| Component Number: $C_{19}$|
| Test Name: Passport Component Testing |
| System: Student Management System |
| Subsystem: Registration |
| Objective: To check the correctness of software component & to calculate the membership value |
| Precondition: User possesses a valid passport |
| Valid for at least 1 year |

**Actions ($A_1 - A_4$)**

- **$SR_{1C_{19}}$:** System displays entered passport number and verifies the passport number validity from the software of Ministry of Interior ✓
  - Membership $m_{1,19} = 0.17$

- **$SR_{2C_{19}}$:** System displays name, father name, date of birth and nationality ✓ (New Recommendation)
  - Membership $m_{2,19} = 0$

- **$A_1$:** If expiry date 2030-12-1

- **$SR_{3C_{19}}$:** System stores the passport expiry date ✓
  - Membership $m_{3,19} = 0.17$

- **$A_2$:** If expiry date 1-2030-12-1

- **$SR_{4C_{19}}$:** Invalid date format "Please enter YYYY-MM-DD" ✓
  - Membership $m_{4,19} = 0.17$

- **$A_3$:** If expiry date 2030-12-1

- **$SR_{5C_{19}}$:** Passport must be valid for at least 1 year ✓
  - Membership $m_{5,19} = 0.17$

- **$SR_{6C_{19}}$:** For already registered students’ system shall send email to renew passport if expiry date of passport is less than 6 months ✓ (New Recommendation)
  - Membership $m_{6,19} = 0$

**Software Component**

- Passport Number
- Issue Date: 00-01-2030
- Expiry Date: 31-12-2030

$P_{C_{19}}^{29} \rightarrow SR_{1C_{19}}, SR_{2C_{19}}, SR_{3C_{19}}, SR_{4C_{19}}, SR_{5C_{19}}, SR_{6C_{19}}$

As $m_{19} = 0.68$, therefore $C_{19} = m_{19} - 0.68$

**Figure 9. Fuzzy test for passport software component.**

This tested passport software component consisted of four expected system responses and two new expected system responses were recommended during the requirements validation process that were successfully tested with system responses $SR_{1C_{19}},$
The $SR_i C_{i9}$, $SR_i C_{19}$, $SR_i C_{19}$, $SR_i C_{19}$, and $SR_i C_{19}$. The $\checkmark$ symbol indicates that system response was successfully tested, whereas the $\times$ symbol indicates that it is new recommendation. If $SR_i C_i$ was successfully performed, then $SR_i C_i$ was assigned one membership value, else a zero membership value was assigned. The fuzzy membership value of this software component was $m_{i9} = 0.68$. The fuzzy membership value was calculated by equation 1 given below.

$$m_i = \sum_{k=1}^{n} m_{k,i} = \sum_{k=1}^{n} \frac{SR_k C_i}{SRC_i}$$

During the validation process, the following changes were required in the software, and we classified them into content requirements and inverse requirements, as shown in Tables 2 and 3. In this study, we measured the completeness of software by using our introduced approach. The software components were designed with single functionality or with sub-functionalities which are cohesive. However, in order to integrate different software components together, software components were loosely coupled.

**Table 2. Content requirements.**

| Description                                                                 | Software Components                                                                 |
|-----------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| $R_1^7$: The system must display a drop-down list for the selection of the course category (professional compulsory course, professional elective course, public basic course, public compulsory course). | ![Course Category](image) Select ▼ ▼ |
| $C_P = 0.81$                                                               | ![Professional Compulsory](image) Courses Code Select ▼ ▼ |
| $R_2^8$: The system must display the selected course category and ask the user to select a course code offered by the university during the current semester. | ![Professional Compulsory](image) Courses Code Select ▼ ▼ |
| $C_P = 0.88$                                                               | ![Professional Compulsory](image) Courses Code Select ▼ ▼ |
| $R_3^{12}$: The system must calculate the CGPA according to the university format and ask the user to enter obtain marks and total marks for each subject. | ![CGPA](image) Enter Obtained Marks Enter Total Marks Add Details for Another Subject Yes No |
| $C_P = 0.87$                                                               | ![Professional Compulsory](image) Courses Code Select ▼ ▼ |
| $R_4^{52}$: The system must display the recommended books for the entered course code. | ![Course Code](image) Select ▼ ▼ |
| $C_P = 0.64$                                                               | ![Professional Compulsory](image) Courses Code Select ▼ ▼ |
| $R_5^{189}$: The system must display a text box to enter a student ID. (The system generates a student card) | ![Student ID](image) Select ▼ ▼ |
| $C_P = 0.91$                                                               | ![Professional Compulsory](image) Courses Code Select ▼ ▼ |
| $R_6^{190}$: The system must display a text box to enter a faculty ID. (The system generates a faculty card) | ![Faculty ID](image) Select ▼ ▼ |
| $C_P = 0.88$                                                               | ![Professional Compulsory](image) Courses Code Select ▼ ▼ |
\( R_{17} \): The system must display a text box to enter a faculty ID and research interest. (The system generates a profile)
\[ C_P = 0.83 \]

\( R_{28} \): The system must display a text box to enter the ID of a degree course. (The system generates a list of courses offered during the current semester)
\[ C_P = 0.94 \]

\( R_{9} \): The system must display a text box to enter a student class ID. (The system generates a timetable for the student)
\[ C_P = 0.92 \]

\( R_{10} \): The system must display a text box to enter a faculty class ID. (The system generates a timetable for the faculty member)
\[ C_P = 0.88 \]

\( R_{11} \): The system must display a text box to enter a student ID and a degree ID. (The system generates a transcript)
\[ C_P = 0.95 \]

Table 3. Inverse requirements.

| Description | Software Components |
|-------------|---------------------|
| \( R_{12} \): The system shall not print a semester invoice after the due date. (This option is for offline payment and bank name will be printed on the semester invoice) \[ C_P = 0.91 \] | Semester Invoice |
| \( R_{2} \): The system shall display the text box in red if a user enters more than or less than eight digits for their national identity card number. \[ C_P = 0.83 \] | NIC Number |
| \( R_{3} \): The system shall display the text box for age in red if the age of a candidate is less than 18 years or more than 35 years. A text box in lighter blue indicates that the age will be automatically calculated. (Red text box indicates the ineligibility of a candidate) \[ C_P = 0.80 \] | Date of Joining, Date of Birth, Age |
| \( R_{4} \): The system shall not accept a native language other than Chinese/English. \[ C_P = 0.89 \] | Native Language |
| \( R_{5} \): The system shall not accept a nationality of a candidate other than that of their country of residence. \[ C_P = 0.91 \] | Nationality |
| \( R_{6} \): The system shall display a red text box if a user enters a percentage of completeness of a master degree that is less than 70%. \[ C_P = 0.88 \] | Master Degree |
The system shall not upload a photograph for a user profile if its size is less than (300 X 400) pixels. 

\[ R_7^{296} : \text{The system shall not upload a photograph for a user profile if its size is less than (300 X 400) pixels.} \]

\[ C_p = 0.73 \]

The system shall not accept journal publications other than those from the SCI/SCIE. 

\[ R_8^{301} : \text{The system shall not accept journal publications other than those from the SCI/SCIE.} \]

\[ C_p = 0.60 \]

The system shall not accept conference publications that are not recognized by the CCF/EC. 

\[ R_9^{402} : \text{The system shall not accept conference publications that are not recognized by the CCF/EC.} \]

\[ C_p = 0.70 \]

The following are the complexity factors affecting developed methodology: size, goals and objectives, changes, ambiguity and data.

Size: The size of the software component is considered as an indication of complexity, and for this software components were designed to be symmetric with a minimum size. Goals and objectives: If goals and objectives are not defined properly, then the tested software component neglects to test the important features. Changes: Changes in the functionality of software components are affected by internal and external influences. They imply adaption to changing requirements. Ambiguity: In order to avoid ambiguity, each software component was validated by a requirements validation framework. Data: Incomplete data fail to test the expected system response. This fuzzy test was designed by considering software requirements and the dataset.

The software consisted of 360 software components, 323 existing software components and 37 new software components. In this study, 21 software components required partial changes, as shown in Table 2 and Table 3, and Figure 9 also represents a software component that required partial changes. All software components were tested by a fuzzy test, to find the \( C_C \), \( C_P \) and \( C_N \) values.

The details regarding the functionalities are as follows: 342 software components consisted of a single functionality, 11 consisted of 2 functionalities and 7 consisted of 3 functionalities. Table 1 illustrates the detail of the dataset. Figures 8 and 9 represent a fuzzy test of the software components \( C_5 \) and \( C_9 \), respectively.

**Rules for Assigning the Fuzzy Membership Value**

- if the fuzzy membership value of \( m_i \) is equal to one, then
  \[ C_C = m_i \]

- else if the fuzzy membership value of \( m_i \) varies between zero and one, then
  \[ C_P = m_i \]

- else
  \[ C_N = m_i \]

end if

\[ C_C = 302 \text{ and } C_P = 0.81 + 0.88 + 0.87 + 0.64 + 0.91 + 0.88 + 0.83 + 0.94 + 0.92 + 0.88 + 0.95 + 0.91 + 0.83 + 0.80 + 0.89 + 0.91 + 0.88 + 0.73 + 0.60 + 0.70 + 0.68 = 17.44. \]

In this study, 37 new software components required development. Therefore, \( C_N = 0 \) for the 37 software components. The total software components became \( C_T = 323 + 37 = 360 \). By substituting the values in Equation (2),

\[ S_i = \left[ 1 - \left( \frac{C_C}{C_T} + \frac{C_P}{C_T} + \frac{C_N}{C_T} \right) \right] \]

\[ S_i = \left[ 1 - \left( \frac{302}{360} + \frac{17.44}{360} + \frac{0}{360} \right) \right] \]
The software obtained a completeness of 0.88, and to obtain the complete membership value, software required a reverse engineering membership value of 0.12. $S_E = 0$ because the existing software was maintained.

7. Results

In the experiment, we observed that if software completeness had a membership value less than a 0.30, then it would be developed by forward engineering. For reverse engineering, existing software components required minor changes, whereas few new software components were developed. As the software size increased, its complexity also increased and resembled an Egyptian pyramid. Figure 10 represents software completeness. Membership is represented on the Y-axis and changes on the X-axis.

In the experiment, the software consisted of three hundred twenty-three existing software components, out of which twenty-one software components were partially changed. To update the software to a new version or change it according to the latest software requirements, twenty-one software components were restructured for new functionalities and features, and thirty-seven software components were newly developed. Therefore, the total number of software components after the reverse engineering process was three hundred sixty. In the figure, 0.88 represents the fuzzy membership value of software completeness. To achieve the complete membership value (one) for the software, new software components were developed, and a few were partially changed. The change in the membership value was 0.12.

To determine whether the software required minor, major or complex changes, we used the linguistic variable: (c, T(c), U, M), as c: “change” is a linguistic variable. T(c) =
{minor changes, major changes, complex changes}, U: U = {0, 360}, M (minor changes) = fuzzy set for change in less than 90 software components with membership of $\mu_{\text{minor changes}}$. In the experiment, there were 37 new software components and 21 software components required partial changes. In this study, the software was developed by component-based development. The details regarding the line of code (LOC) for the software components are as follows: 342 software components were developed with a single functionality which consisted of approximately 32 LOCs per software component, 11 software components were developed with two functionalities which consisted of approximately 60 LOCs per software component and seven software components were developed with three functionalities which consisted of approximately 105 LOCs per software component. The total LOC of the complete software was 12,439.

We validated the software components by the requirements validation framework described in Section 2.1. To validate the introduced approach, we applied a machine learning confusion matrix, which is used by a prediction model to store incorrect and correct decisions [62]. Software modification means changes required in the software components according to new software requirements or due to errors or defects in the software components. If a software component is classified as correct when it is truly correct, then the classification is a true positive (TP). If the software component is classified as correct when it is actually modified, then the classification is a false positive (FP). If the software component is classified as a modification when it is actually correct, then the classification is a false negative (FN). If the software component is classified as a modification when it is actually modified, then the classification is a true negative (TN).

In this study, the total number of predictions regarding the software component changes (partial membership) was 360 and there were 352 correct predictions; true positive (TP) = 322, false positive (FP) = 6, false negative (FN) = 2, true negative (TN) = 30.

- Accuracy = $\frac{TP + TN}{Total} = \frac{322 + 30}{360} = 0.97$
- Sensitivity = Recall = $\frac{TP}{FN + TP} = \frac{322}{2 + 322} = 0.99$
- Specificity = $\frac{TN}{TN + FP} = \frac{30}{30 + 6} = 0.83$
- Precision = $\frac{TP}{TP + FP} = \frac{322}{322 + 6} = 0.98$

In this research, we used a machine learning technique for measuring the accuracy of the introduced approach, which is 0.97.

8. Conclusions

In this study, software completeness was measured with a quantitative approach. Software completeness scaled the membership value between zero and one. Values closer to one indicated that minor modifications or design recoveries were required by reverse engineering. If the value was closer to zero, then the software was developed by forward engineering. Each software component was tested by a fuzzy test. The fuzzy test extracted the errors and missing features within software components. The software components with a partial membership value were easily maintained due to the identification of an error from the exact line of code. The mathematical formula of $S_c$ was proved with three conditions: complete membership, partial membership and non-membership. The classification of software requirements helped to developed software components with improved quality, and these software components were also reliable, upgradable and reusable. The introduced approach developed high-quality software, which reduces the development cost and as well as the testing cost. It has been observed that after two to three years, in only a few circumstances, software fulfills the demanded requirements and possesses a software completeness value equal to one. However, in most circumstances, software exhibits partial membership.

In future work, we are interested to measure fuzzy maintenance cost and fuzzy maintenance line of code based on the number of errors or defects identified by the fuzzy test.
Author Contributions: Conceptualization, N.I. and J.S.; Data curation, N.I.; Methodology, N.I. and J.S.; Supervision, J.S.; Writing—original draft, N.I.; Writing—review and editing, J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Chongqing Graduate Tutor Team Construction Project, grant number ydstd1821.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The fuzzy tests generated during the study for testing the software components are openly available, at: https://github.com/Nayyar-Iqbal/Fuzzy-Software-Completeness.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

\( S_F \): Set of features
\( R \): Requirements
\( SR_{ki} \): System response tested for the \( i \)th software component \( C_i \) that contains \( k \) number of system responses
\( SRC_i \): Total system response tested for the \( i \)th software component \( C_i \)
\( FR \): Functional requirements
\( NFR \): Non-functional requirements
\( IR \): Inverse requirements
\( NIR \): Non-inverse requirements
\( CR \): Content requirements
\( NCR \): Non-content requirements
\( S \): Software
\( C_i \): Software component, where \( i \) represents software component number, \( i = \{1, 2, 3, ..., n\} \) or \( i = \{a, b, c, ..., z\} \)
\( R_{ji} \): Represents the \( j \)th software component, where \( j \) represents the software requirement number
\( FC_{ki} \): Represents the functionality of the component, \( i \) represents the software component and \( k \) represents the functionality number of the \( i \)th software component
\( M_f \): Module, \( f \) represents module number
\( M_p \): Module, \( p \) represents module number
\( \in \): Belongs to
\( \triangleq \): Corresponds to
\( SE \): Engineering of software
\( C_C \): Complete membership of the software components
\( C_P \): Partial membership of the software components
\( C_N \): Non-membership of the software components
\( C_T \): Total number of software components
\( S_i \): Software components completeness index
\( S_C \): Software completeness
\( m_{ki} \): Membership value of the software response for software component \( C_i \), \( k \) response number
\( m_i \): Fuzzy membership value (\( 0 \leq m_i \leq 1 \))
\( P_S \): Software program
\( TP \): True positive
\( FP \): False positive
\( FN \): False negative
\( TN \): True negative
References

1. Cabrera, I.P.; Cordero, P.; Ojeda-Aciego, M. Fuzzy logic, soft computing, and applications. In Proceedings of the 10th International Work-Conference on Artificial Neural Networks, Salamanca, Spain, 10–12 June 2009; pp. 236–244.

2. So, S.; Cha, S.; Kwon, Y. Empirical evaluation of a fuzzy logic-based software quality prediction model. Fuzz Sets Syst. 2002, 127, 199–208.

3. Bosc, P.; Damiani, E.; Fugini, M. Fuzzy service selection in a distributed object-oriented environment. IEEE Trans. Fuzz Syst. 2001, 9, 682–698.

4. Idri, A.; Abran, A.; A fuzzy logic based measures for software project similarity: Validation and possible improvements. In Proceedings of the 7th International Software Metrics Symposium, London, UK, 4–6 April 2001; pp. 85–96.

5. Pang, C. Software Engineering for Agile Application Development; IGI Global Publisher of Timely Knowledge; Seveco AG: Uster, Mönchaltorf, Switzerland, 2020.

6. Davis, A.; Overmyer, S.; Jordan, K.; Caruso, J.; Dandashi, F.; Dinh, A.; Kincaid, G.; Ledeboer, G.; Reynolds, P.; Sitaram, P.; et al. Identifying and measuring quality in a software requirements specification. In Proceedings of the First International Software Metrics Symposium, Baltimore, MD, USA, 21–22 May 1993; pp. 141–152.

7. Iqbal, N.; Sang, J.; Gao, M.; Hu, H.; Xiang, H. Forward engineering completeness for software by using requirements validation framework. In Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering, Lisbon, Portugal, 10–12 July 2019; KSI Research Inc.: Pittsburg, PA, USA, 2019; pp. 523–528.

8. Laptante, P.A. Requirements Engineering for Software and System, 3rd ed.; CRC Press/Taylor & Francis: Boca Raton, FL, USA, 2017; pp. 15–45.

9. Menzel, I.; Mueller, M.; Gross, A.; Doerr, J. An experimental comparison regarding the completeness of functional requirements specifications. In Proceedings of the 18th IEEE International Requirements Engineering Conference, Sydney, NSW, Australia, 27 September–1 October 2010; IEEE: New York City, NY, USA, 2010; pp. 15–24.

10. Xie, G.; Chen, J.; Neamtuu, I. Towards a better understanding of software evolution: An empirical study on open source software. In Proceedings of the IEEE International Conference on Software Maintenance, Edmonton, AB, Canada, 20–26 September 2009; pp. 51–60.

11. Robbes, R.; Lanza, M. A changed-based approach to software evolution. Electron. Notes Theor. Comput. Sci. 2007, 166, 93–109.

12. Robbes, R.; Lanza, M.; Lungu, M. An approach to software evolution based on semantic change. In Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering, Held as Part of the Joint European Conference on Theory and Practice of Software, Braga, Portugal, 24 March–1 April 2007; pp. 27–41.

13. Zave, P. An experiment in feature engineering. In Programming Methodology; Springer: Berlin/Heidelberg, Germany, 2003; pp. 353–377.

14. Graham, D.; Black, R.; Veenendaal, E.V. Foundations of Software Testing, 4th ed.; Cengage Learning EMEA: Somerset West, UK, 2019.

15. Patton, R. Software Testing, 2nd ed.; Sams Publishing: Carmel, IN, USA, 2005.

16. Homès, B. Fundamentals of Software Testing; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2013.

17. Jørgensen, P.C. Software Testing: A Craftsman’s Approach, 4th ed.; Auerbach Publications: Boca Raton, FL, USA, 2013.

18. Naik, K.; Tripathy, P. Software Testing and Quality Assurance: Theory and Practice; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2008.

19. Apel, S.; Kastner, C.; Batory, D. Program refactoring using functional aspects. In Proceedings of the 7th International Conference on Generative Programming and Component Engineering, Nashville, TN, USA, 19–23 October 2008; pp. 161–170.

20. Liu, W.; He, C.; Zhang, K. Service-based domain requirements completeness analysis. In Proceedings of the Asia-Pacific Conference on Computational Intelligence and Industrial Applications; Wuhan, China, 28–29 November 2009; pp. 110–115.

21. Bezdek, J.C. Fuzzy models-what are they, and why? IEEE Trans. Fuzz Syst. 1993, 1, 1–6.

22. Bandler, W.; Kohout, L.J. Fuzzy power sets and fuzzy implication operators. Fuzz Sets Syst. 1980, 4, 13–30.

23. Zadeh, L. Fuzzy sets. Inf. Control. 1965, 8, 338–353.

24. Sleimi, A.; Sannier, N.; Sabetzadeh, M.; Briand, L.C.; Dann, J. Automated extraction of semantic legal metadata using natural language processing. In Proceedings of the 26th IEEE International Requirements Engineering Conference, Banff, AB, Canada, 20–24 August 2018; pp. 124–135.

25. Brooks, R. Data Privacy: What You Need to Know and How to Get Started. Available online: https://blog.netwrix.com/2019/08/08/data-privacy/ (accessed on 10 February 2021).

26. Wegener, J.; Grochtmann, M. Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. Real Time Syst. 1998, 15, 275–298.

27. Iqbal, N.; Sang, J.; Chen, J.; Xia, X. Measuring Software Maintainability with Naïve Bayes Classifier. Entropy 2021, 23, 136, doi:10.3390/e23020136.

28. Ansari, M.Z.A. Softrology: Learn Software Technologies. Available online: https://softrology.blogspot.com/search?q=inverse (accessed on 25 November 2020).

29. Zadeh, L.A. The concept of a linguistic variable and its application to approximate reasoning-I. Inf. Sci. 1975, 8, 199–249.

30. Kacprzyk, J.; Yager, R. Linguistic summaries of data using fuzzy logic. Int. J. Gen. Syst. 2001, 30, 133–154.

31. Murray, T. Proofs and Side Effects. Available online: https://verse.systems/blog/post/2018-10-02-Proofs-And-Side-Effects/ (accessed on 11 February 2021).
32. Cadar, C.; Dunbar, D.; Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, San Diego, CA, USA, 8–10 December 2008; pp. 209–224.

33. Beurdouche, B.; Lavaud, A.D.; Kobeissi, N.; Pironti, A.; Bhargavan, K. FLEXTLS: A tool for testing TLS implementations. In Proceedings of the 9th USENIX Workshop on Offensive Technologies, Washington, DC, USA, 10–11 August 2015; pp. 1–10.

34. Teplyuk, P.A.; Yakunin, A.G.; Sharlaev, E.V. Study of security flaws in the Linux kernel by fuzzing. In Proceedings of the 2020 International Multi-Conference on Industrial Engineering and Modern Technologies, Vladivostok, Russia, 6–9 October 2020; pp. 1–5.

35. Bhargavan, K.; Fournet, C.; Kohlweiss, M.; Pironti, A.; Strub, P. Implementing TLS with verified cryptographic security. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 19–22 May 2013; pp. 445–459.

36. MacDonell, S.G.; Gray, A.R. Applying fuzzy logic modeling to software project management. Softw. Eng. Comput. 2003, 731, 17–43.

37. Boehm, B.W. Software Engineering Economics, IEEE Trans. Softw.Eng. 1984, SE-10(1): 4-21.

38. Gustafson, D. Schaum’s Outlines Software Engineering, 1st ed.; McGraw-Hill Education: New York, NY, USA, 2002.

39. Bano, M.; Zowghi, D. A systematic review on the relationship between user involvement and system success. Inf. Softw. Technol. 2015, 58, 148–169.

40. Pagano, D.; MaaJe, W. User feedback in the appstore: An empirical study. In Proceedings of the IEEE 21st International Requirements Engineering Conference, Rio de Janeiro-RJ, Brazil, 15–19 July 2013; pp. 125–134.

41. Carreno, L.V.G.; Winbladh, K. Analysis of user comments: An approach for software requirements evolution. In Proceedings of the 35th International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 582–591.

42. Pagano, D.; Bruegge, B. User involvement in software evolution practice: A case study. In Proceedings of the 35th International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 953–962.

43. Sram, N. Practical application of fuzzy logic: Experiences and current state in software engineering. In Proceedings of the 12th IEEE International Symposium on Computational Intelligence and Informatics, Budapest, Hungary, 21–22 November 2011; pp. 579–581.

44. Yadav, H.B.; Yadav, D.K. A fuzzy logic based approach for phase-wise software defects prediction using software metrics. Infor. Softw. Tech. 2015, 63, 44–57.

45. Al-Refaei, M.; Cazzola, W.; Ghosh, S. A fuzzy logic based approach for model-based regression test selection. In Proceedings of the 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and systems, Austin, TX, USA, 17–22 September 2017; pp. 55–62.

46. Mittal, A.; Parkash, K.; Mittal, H. Software cost estimation using fuzzy logic. ACM SIGSOFT Softw. Eng. Notes 2017, 35, 1–7.

47. Yadav, D.K.; Dutta, S. Test case prioritization technique based on early fault detection using fuzzy logic. In Proceedings of the 3rd International Conference on Computing for Sustainable Global Development, New Delhi, India, 16–18 March 2016; pp. 1033–1036.

48. Idrï, A.; Abran, A.; Khoshgoftaar, T.M. Fuzzy case-based reasoning models for software cost estimation. Soft Comput. Softw. Eng. 2004, 159, 64–96.

49. Nisar, M.W.; Wang, Y.J.; Elahi, M.; Software development effort estimation using fuzzy logic—A survey. In Proceedings of the 2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery, Shandong, China, 18–20 October 2008; pp. 421–427.

50. Kataev, M.; Bulysheva, L.; Xu, L.; Ekhlakov, Y.; Permyakova, N.; Jovanovic, V. Fuzzy model estimation of the risk factors impact on the target of promotion of the software product. J. Enterp. Inf. Syst. 2020, 14, 797–811.

51. Lima, D.C.; Freitas, F.; Campos, G.; Souza, J. A fuzzy approach to requirements prioritization. In Proceedings of the 3rd International Symposium on Search Based Software Engineering, Szeged, Hungary, 10–12 September 2011; pp. 64–69.

52. Konar, A. Fuzzy pattern recognition. In Computational Intelligence; Springer: Berlin/Heidelberg, Germany, 2005; pp. 125–138.

53. Chang, C.L. Fuzzy-Logic-Based Programming; World Scientific Publishing: Singapore, 1997.

54. Jurado, F.; Redondo, M.A.; Ortega, M. Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice. J. Netw. Comput. Appl. 2012, 35, 695–712.

55. Taratuknin, V.; Yadgarova, Y. A fuzzy logic approach for product configuration and requirements management. In Proceedings of the 2015 Annual Conference of the North American Fuzzy Information Processing Society held jointly with 2015 5th World Conference on Soft Computing, Redmond, WA, USA, 17–19 August 2015; pp. 1–5.

56. Pressman, R.S. Software Engineering: A Practitioner’s Approach, 8th ed.; McGraw-Hill Education: New York, NY, USA, 2014.

57. Young, R.R. Recommended requirements gathering practices. J. Def. Softw. Eng. 2002, 15, 9–12.

58. Zowghi, D.; Coulin, C. Requirements elicitation: A survey of techniques, approaches, and tools. In Engineering and Managing Software Requirements, 1st ed.; Aurum, A., Wohlin, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 19–46.

59. Duggan, E.W.; Thachenkary, C.S. Integrating nominal group technique and joint application development for improved systems requirements determination. Inf. Manag. 2004, 41, 399–411.

60. Coughlan, J.; Lycett, M.; Macredie, R.D. Communication issues in requirements elicitation: A content analysis of stakeholder experiences. Inf. Softw. Technol. 2003, 45, 54–61.
61. Peruma, A.; Almalki, K.; Newman, C.D.; Mkaouer, M.W.; Ouni, A.; Palomba, F. Tsdetect: An open source test smells detection tool. In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, New York, NY, USA, 8–13 November 2020; pp. 1650–1654.

62. Brownlee, J. What Is a Confusion Matrix in Machine Learning. Available online: https://machinelearningmastery.com/confusion-matrix-machine-learning/ (accessed on 15 December 2020).