International Journal of Innovative Research in Computer Science & Technology (IJIRCST)
ISSN: 2347-5552, Volume-9, Issue-2, March 2021
https://doi.org/10.21276/ijircst.2021.9.2.3
Article ID IRP1127, Pages 14-21
www.ijircst.org

Fuzzy Logic Support for Requirements Engineering

Ahmet Egesoy[1], and Aylin Güzel[2]

[1] Assistant Professor, Department of Computer Engineering, Ege University, İzmir, Turkey
[2] PhD Student, Department of Computer Engineering, Ege University, İzmir, Turkey

Correspondence should be addressed to Ahmet Egesoy; ahmet.egesoy@ege.edu.tr

ABSTRACT- As software projects are getting more and more complicated, the greatest risks in software engineering are most probably emanating from the limitations of an inexperienced developer in imagining the boundaries of an abstract artefact that does not even exist yet. Requirement engineering is extremely important in a software development project, yet inherently difficult. Requirements can be redundant, optional, overlapping and even contradictory. They come from different sources and often are represented in an informal way. Requirements also are followed throughout the development process and can be partially met in different degrees in various stages of the process. In this work we advocate a fuzzy logical model for following the requirements and their fulfillment. We also present the logical design of a requirements knowledge base manager that we are building in order to facilitate requirement-aware rapid development tools.

KEYWORDS- Artificial Intelligence, Fuzzy Logic, Requirement Engineering, Software Engineering.

I. INTRODUCTION

Requirement engineering is probably the most important task in a software development project. Requirements come from multiple stakeholders that have distinct interests, intensions and world views. This makes requirement specification a difficult task not only because it demands effective communication skills but also because requirements can be interrelated in unexpected ways.

In the basic software development tradition which was later known as the waterfall process model, requirement engineering naturally is arranged to be the first phase of the development process. Later development methods that adopt various forms of repetitive improvement, assume that requirements engineering continues through a system's lifetime. As the system evolves towards completion requirements are gradually met. Some of them may remain in a semi-finished condition for a long time.

Structured methods that deal with functional requirements often arrange them in a hierarchy. In this approach it is also possible to differentiate optional requirements from mandatory requirements and form a meta-model of acceptable system configurations. The requirements of the system are either fulfilled or not. However, requirements can often be difficult to be captured by a rigid structure. Instead they can often be redundant, incomplete, fuzzy, overlapping and even contradictory.

In the second section, the existing literature of the software engineering related usage of fuzzy logic is summarized. The third section contains some theoretical remarks about fuzzy logic and where and why we need it. The forth section gives an outline of our *Requirements Knowledge Base Manager* project, without going into technical details unless it is for highlighting some interesting points regarding our original approach to fuzzy logic. The fifth section contains conclusions.

II. LITERATURE OF FUZZINESS IN REQUIREMENTS ENGINEERING

The current software development literature clearly indicates the crucial role of requirements engineering in the software development life cycle. There are cases where a fuzzy logic approach has been employed in order to prioritize requirements. Software defects can also be detected with fuzzy logic. Fuzzy logic is also found to be used for the estimation of cost, risk, reliability or total development effort that will be necessary. Fuzzy logic was also used for developing an intelligent recommendation system that can hunt the requirements in the informal definitions of large-scale software projects.

The foremost motivation in requirement specification is to prevent any of the indispensable requirements to be omitted. Chakraborty et al. [1] in their study emphasized that requirements engineering was the most important stage in the software development life cycle. They also stated that, this stage was used to transform the missing needs and requests of potential software users into complete, precise and formal features.

Burgin et al. [2] in their study, tried to show how uncertainty arises in software engineering and how this uncertainty can be reflected in measuring software qualities. They also emphasized that software measurement plays a critical role in all stages of the software life cycle.

Lima et al. [3] investigated, one of the most important issues related to the efficiency of software development which is prioritizing of the fulfilment of requirements. In this work requirement prioritization was found to have ambiguous aspects, so that fuzzy logic concepts have been advocated to represent and solve the problem much more accurately.

Yadav et al. [4] proposed a model that predicted the number of design defects before the test phase. In this work,

software metrics are taken into account to develop models for early software error prediction. Software size metric and requirement analysis results are used to predict the number of possible defects during testing using fuzzy logic rules. In this study, 20 real software project data sets are used to show the validity and usability of the proposed approach.

In the work of Huang et al. [5] a new neuro-fuzzy cost model has been proposed for software cost estimation. The model carries some characteristics of the *neuro-fuzzy approach*, such as learning ability and good interpretability.

Nisar et al. [6] focused on effort estimation that aims at estimating the number of work hours and workers that are needed to develop a project. The purpose of their research has been to analyze the use of fuzzy logic in existing cost estimation models and to examine, in-depth software and project estimation techniques available in the industry, with an assessment of their strengths and weaknesses.

A fuzzy model by Aljahdali et al. [7] was developed to estimate the reliability of software projects. The Takagi-Sugeno technique was used in their own fuzzy models. In this study fuzzy models were tested in three kinds of applications which are real-time control systems, military applications and operating systems.

A new recommendation system based on *Apriori algorithm* was proposed by Alzu'bi et al. [8] for recommending user requirements. In this study, a data set containing 4000 records was used. User-created rules are activated for analyzing informal text data in order to suggest requirements to users.

Bubenko et al. [9] in their work, summarized the *expanded requirements and information modeling paradigm* based on their interrelated meta-model. In this study, requirements engineering was defined as: the *systematic process of developing requirements to analyze the problem, document the resulting observations and check their accuracy*. This ambitious work focused on the reuse process and emphasized the necessity of having a reuse engineer responsible for design in the development process.

Yegorov et al. [10] studied the use of T-norm functions which define fuzzy intersection of sets and conjunction in fuzzy logic. They also tried to demonstrate how these functions can be used in requirement specification. They emphasized the verification of requirements during the acceptance tests of information systems. They explained that the purpose of verification was to determine the quality of the software product by checking the software's compliance with functional and non-functional requirements.

Ramzan et al. [11] highlights some serious shortcomings of current requirement prioritization techniques. In this study, an intelligent fuzzy logic-based technique is proposed for requirement prioritization. The technique uses fuzzy logic to prioritize requirements based on their *perceived value*. It is recommended that the system creates two separate requirement documents in order of priority. The first document containing requirements that were given higher priority than a certain threshold while the second list being the requirements that have a lower priority than the specified threshold. Their approach was basic and they emphasized that any requirement prioritization technique would gain a much wider acceptance if it was easy to use and understand.

The work by Ebraert et al. [12] has tried to bridge the design and implementation phases by using *change-based FODA diagrams* for product line engineering. Software product line engineering is a software engineering paradigm that encourages reuse throughout software development. The role of the FODA diagram is to briefly describe which feature combinations are allowed in the system. FODA diagrams are considered as a mechanism to fill the gap between requirements and design. In this approach FODA diagram has been shown to be useful not only for bridging requirements and design, but also bridging design and implementation. In their study, they proposed a method that can be used to automatically create a FODA diagram from changes in the source code. Benavides et al. [13] also worked on product line engineering. They emphasized that software product line engineering had proven to be an effective method for software production. They advocated feature modeling is to identify similarities and differences between all products of a software product line. Feature models are used to model the software product line in terms of features and relationships between them.

Goncalves et al. [14] focused on the semantic side of fuzziness. According to their work the adjective "*easy*" is an indefinite, thus a fuzzy term, since *convenience* depends on user preferences. An easy course can be defined as a course where all students get high marks. Height (high grade) is an uncertain term, which also makes it *fuzzy*. The authors emphasized that their goal was to provide automated software engineering tools to develop applications with fuzzy requirements. In their study, a method is proposed to develop applications that support fuzzy requirements.

After emphasizing the importance of requirement analysis Liu [15] describes a way for analyzing functional requirements in terms of inputs, outputs and their relationships. They complained that in many software development projects, requirements were sometimes not specified in detail and this made software verification and maintenance very difficult. The main difficulty was that many product requirements were inherently fuzzy. Customers often defined requirements in fuzzy terms such as "good", "high", "low" or "very important".

Hsieh et al. [16] claimed that software development is inherently cursed with complexity, uncertainty and risk. Risk analysis is the most critical activity in a software project, but risk assessment is often under-done. Managers need more effective tools to reduce the high failure rate of software projects. Fuzzy logic is well suited for analysis in this case. Risk assessment steps are:

- Information is collected.
- Risks are defined.
- Membership functions are defined. (For fuzzy logic)
- Risks are rated. (extreme high, very high, high, quite high, medium, quite low, low)
- Weights are evaluated. (Fuzzy logic)
- Risk assessment is completed.

This fuzzy logic based risk assessment technique claimed better accuracy, reducing calculation time and less errors.

III. WHERE TO ALLPY FUZZY LOGIC

A. Fuzzy Logic Systems

Fuzzy logic is a general computation system which is inspired by human thinking. This system is based on the relations between linguistic variables and logical expressions. Fuzzy logic is considered to be an artificial

intelligence subfield. Fuzzy logic is based on the fuzzy set theory. Fuzzy sets are sets with vague boundaries. The membership test to a fuzzy set can return any value between true and false.

While in classical logic we can denote an event with *0 or 1*, in fuzzy logic we can denote infinite values between 0 and 1. In fuzzy logic, a proposition has a certain degree of truth or falsehood. A proposition can be true or false to a certain extent such that it may be *true, false, slightly true*, or *slightly false*.

Fuzzy logic is currently used in many areas of daily life. It reduces waiting time by evaluating passenger traffic in elevator inspection. It determines the best focus and illumination when the cameras have several objects on their visor. In washing machines, fuzzy control systems sense the dirtiness and weight of the laundry as well as the type of the fabric and automatically select the most suitable washing program. Fuzzy logic senses the condition of the surface and the environment in vacuum cleaners and adjusts the engine power accordingly. It adjusts the heating in water heaters according to the amount and temperature of the water used. It detects the best working situation by evaluating the ambient conditions in air conditioners, and increases the cooling power when someone enters the room. Fuzzy software adjust the screen contrast, brightness and color on smart television sets by assessing lighting condition.

A simple fuzzy system design is demontrated in Fig 1. In the *fuzzification* stage, real numeric values are converted into membership values to fuzzy sets. Then, through the *Fuzzy Inference Engine*, IF-THEN type rules are transformed into a fuzzy relationship defined between the input and output space. In the *defuzzification* stage, the fuzzy set is transformed back into the values of the real world. This way a sharp membership output is provided. The defuzzification of values are usually computed by taking weighted averages.

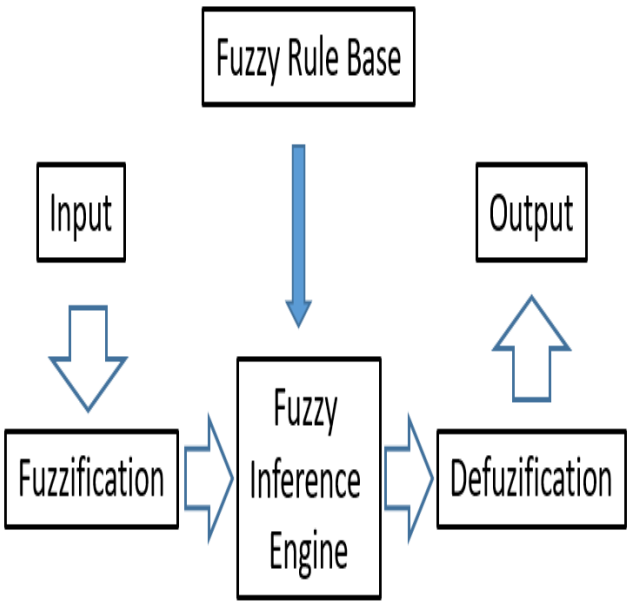

Fig. 1: Design of a basic fuzzy system.

B. Why Use Fuzzy Logic

Requirements analysis is the first stage of the software development process which is critically important. The most dangerous risk in a software project is developing the wrong product. Requirements management should be done accurately and effectively.

Physical conditions, laws, competition conditions and often the wishes of customers are effective in determining the requirements. Unnecessary and duplicate requirements can prevent the scope and cost of the project from being determined. Requirements should be written as clearly as possible in natural language sentences. Performing the wrong requirement analysis can lead to making wrong decisions and even bring the project to a halt. Wrong decisions taken at the stage of requirements determination or early stages of design are more expensive than expected to be corrected later. In order to prevent such unwanted situations, the requirement analysis should be done accurately and efficiently, and the necessary decisions should be made as early as possible and with less cost.

In order to determine the exact areas where AI should be used in requirements engineering, one needs to imagine an interactive development environment that is supported by a decision support system that has access to an information base about the project. There are certain types of information that a developer could request from such an information base.

In this context, we can envisage that the developer may either be trying to determine certain aspects of the current state of the project or she may also be trying to decide which tasks she should prioritize for the most efficient development process.

It should also be noted that planning of software development is very dynamic task since rapid prototyping applied together with small cycles with validation and verification is a very popular method (called the evolutionary software development process) aiming to decrease the risks in developing large and complicated programs. In such a flexible process model some tasks naturally may be in a semi-finished state and modules can also be in a semi-functioning state. These are both potential fuzzy values, and can also lead to other fuzziness such as *relative cost* or *risk*. Computing fuzzy truth values for software however is not a straightforward task since software has some peculiarities when compared with physical artefacts.

The employment of fuzzy logic in requirement engineering also necessitates adopting a strategy for dealing with some inherent irregularities of software requirements potentially from different sources. Firstly, software is an abstract artefact that can be structured in various forms. In an object oriented design for example there can be more than one way a project can be divided into classes and interfaces. Stakeholders may have completely different ideas about the overall architecture of the project and its components. This may create a feeling of incompatibility between the requirements and even contradictions. There can be containment, overlap or dependency relation between modules rendering conventional fuzzy models incomplete. A common mistake is to assume that developers are simply developing the requirements one at a time. Unfortunately coding (or designing) tasks and

requirements are totally different concepts separated with an abstraction layer. A task may fulfill more than one requirement whereas a requirement may well be distributed over multiple tasks. Sometimes a requirement such as a *coding standard* or *naming convention* may be distributed over the whole project just as it is in some crosscutting aspects. Tasks may also arise dynamically and may take the form of modification instead of development from scratch.

Some requirements such as *performance requirements* are very easy to be expressed as fuzzy logic statements since they are quantifiable. The only necessary task is to compare the numeric data with predefined fuzzy sets. Some other requirements that are written in plain text use some fuzzy terms that can be used when creating formal fuzzy requirements. Fuzzy terms such as big, small, short, long, important, very important, are used in order to express expert knowledge in the form of fuzzy rules already. These should only be checked for inner consistency.

As an example, we can imagine defining a requirement such as *"Authorization check is made for logging into the system"*. Let us focus on how such a requirement could be integrated into a fuzzy context. A good idea is to inject some fuzzy terms in order to fuzzify the requirement specification such as: *"It is very important to be authorized to log into the system."* When stated this way the requirement specification not only makes a statement about authorization being a sub-module of *logging into* the system, but it makes a meta-statement about how important that is. In fact, it is also an indirect way of saying that in fact the system could do without *authorization check*, but (although possible) that would be a very unpleasant alternative that receives low validity values.

IV. FUZZY KNOWLEDGE BASE FOR REQUREMENTS MANAGEMENT

A dynamically managed set of requirements necessitates a requirement manager that interacts with the integrated development environment during the cycles of the evolving project. In Fig 2. a context model for our *requirement manager* has been presented. The figure is a UML diagram with some data flow arrows added. It is evident from this architecture that the Requirement Manager plays a very central role in the development.

Every project is stored in the Configuration Manager together with a set of requirements and a set of tests. As the tests are performed some feedback data is generated so that the fulfillment of the requirements can be monitored. During the computations an *Ontology Base* is also employed in order to provide contextual information about the relations between the input of fuzzy operations. Ontological knowledge, just as any reusable entity is being kept in the *Reuse Repository* and regularly updated by the Requirement Manager.

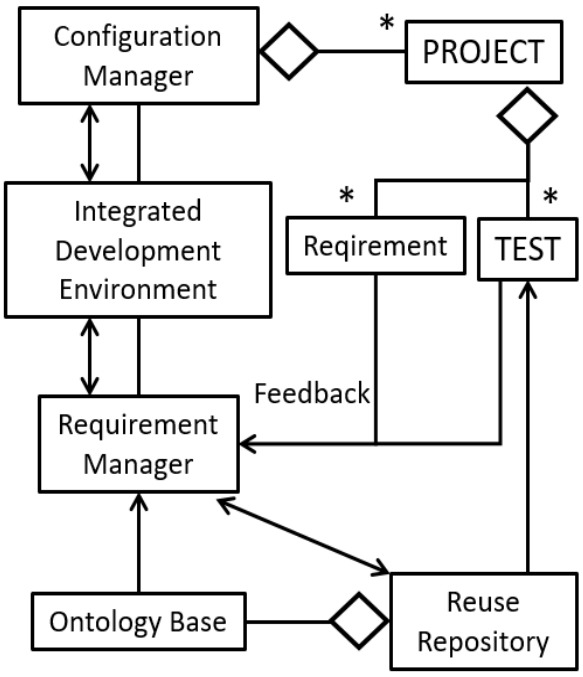

Fig. 2: Context model of requirement manager

At the center of the whole architecture the *Integrated Development Environment* controls the whole process. Code is developed and tested here interacting with the programmer. Information about the current state of concerns is provided continuously.

Our fuzzy logic approach is based on our original unified fuzzy conjunction operator [17] and a fuzzified version of Benlap's truth values (logical constants of Benlap's logic) [18]. Which together form a rather complicated couple, operational details of which is beyond the scope of this paper.

In order to represent the requirements, we structure the fuzzy operations around a fuzzy generalization of *FODA* diagrams [19]. *FODA* stands for *Feature Oriented Domain Analysis*, and *FODA* diagrams are very useful for defining the logical variability of a system by showing the composition of features as a hierarchy where the join points are logical operators. The notation also employs means for expressing mandatory and optional nodes.

A serious attempt for a semi-automated requirements management in a real project not only creates fuzzy values but also has to work with incomplete or inconsistent data. The truth values in such environment has to be fuzzier than standard fuzzy values.

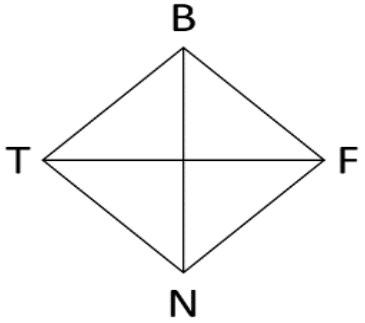

Fig. 3: Benlap's truth values [18]

Fig 3 shows the four logical constants in Benlap's logic. The two values on the left and right sides are *True (T)* and *False (F)* respectively. *B* indicates that *Both true* and *false* hold while *N* indicates that *Neither* of them can be claimed to be *True*. So the values are *True, False, Both and Neither*.

In this respect "True" means that True has been satisfied and False has not been satisfied. In the same manner "False" means that False has been satisfied while True has not been satisfied. Up to this point it is same as classical logic. However, the value B claims that both True and False are satisfied at the same time. While N claims that neither True nor False are satisfied.

In fact, fuzzy generalization of these values can often be easier to understand. In this case, not just the values at the corners of the diamond but the points all over the area of the diamond are considered as valid values of a logical variable. The line between *T* and *F* represents the classical fuzziness. The area below that line is where the variable is under-constrained (has some freedom to change its value). This representation introduces a dimension of *intuitionism* into the value and the value is composed of three real numbers for the degree of being *True*, *False* or *Unknown*. On the other hand, the area above the line represents the cases where the variable in question is over-constrained. In this case the dimension of *paraconsistency* (*True* and *False* coexisting) has been integrated with the value and third value along with *True* and *False* represents the amount of Contradiction. As a result, the truth value is two dimensional; one indicating the position on the *True-False* axis and the other on the *Contradiction-Unknown* axis.

Two dimensional fuzzy truth values allow us to have more complicated *intensions* over the tree of features. It is possible for the queries to conditionally *play safe* or *take risks*. A conventional query that is sent to a conventional knowledge base would have a single intension that can be represented with a discrete *True*. The query has to return a value that is True; and that is the one and only criterion for successful operation of a discrete knowledge base. Two dimensional fuzzy truth on the other hand can aim *precision*, *consistency*, *optimism* or *pessimism* when it gets necessary. For example, the unknown component of a fuzzy variable can be *grounded* as *True* or *False* based on the current degree of *optimism/pessimism* in the computation. This *mood* parameter can be manipulated in a local rule or can be a global environment variable that is stored within a package.

To illustrate our case, let us think of a similar situation about a Computer Aided Diagnosis system that employs fuzzy logic. Such systems are used for tasks as serious as cancer diagnosis, therefore the worst outcome that should be avoided at any cost is a *false negative*, which means that the expert system has diagnosed the patient as *healthy* while he had cancer. We do not want that deadly error, so we would like the system to lean towards the value *True* (cancer positive) in the case of any doubt. (technically called *optimist* because of leaning towards *True*, although semantically it is a pessimistic point of view). It is also possible to make a system for example 0.7 *optimist* or make the system lean diagonally for example towards *False* and *Unknown* at the same time. This generalization of fuzzy logic is advantageous in simulating humanistic priorities when computing in a real life environment.

Another addition we have made to the classical fuzzy logic is in the implementation of fuzzy logic operators. In fuzzy logic *fuzzy t-norm* and *t-conorm* functions are used for computing the output of logical operators (*AND* and *OR* respectively). These functions are said to be *truth-functional* which means that they do not require any input other than two truth values. However, as we have discussed in [16], a realistic computation of real life fuzzy values requires contextual knowledge about the exact semantic relation between the two (or more) operators.

For this reason, it is our view that a plain FODA tree is not sufficient for representing requirement fulfillment computations even though the leaves of the tree are supplied with fuzzy values. The tree has to be augmented by adding associations that indicate the semantic condition between the branches (horizontal links).

As shown in the UML diagram in Fig 4. the concerns of the requirement manager are derived from an *augmented tree*. A *concern* is a value that is monitored by the system during the development process. This value is computed from a number of other parameters by following guidelines that are defined as augmented trees. The system has three built in concerns: *Finishedness*, *Functionality* and *Quality*.

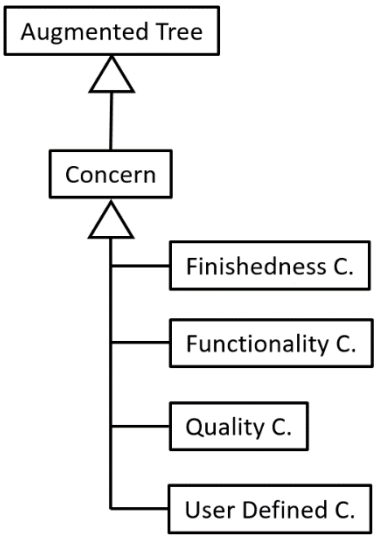

Fig 4: Concerns in Requirement Manager

Finishedness Concern shows the total effective effort that has been spent for the project with respect to the total effort required to finish the project. *Functionality Concern* represents the degree to which the software is performing its function. *Quality Concern* computes the overall quality of the artefact by using predefined quality measures, including code quality.

A forth type of concern is the *User Defined Concern*. The user may add as many of these as she likes. Some of these can involve parameters from the whole project; the others can be concerns about some local parameters. The developer can add concerns for certain specific types of functionalities such as "user interaction" or "connectivity". The functionalities of specific subsystems can also be watched. For example, the developer may want to know up to what degree the *data logging subsystem* is functional.

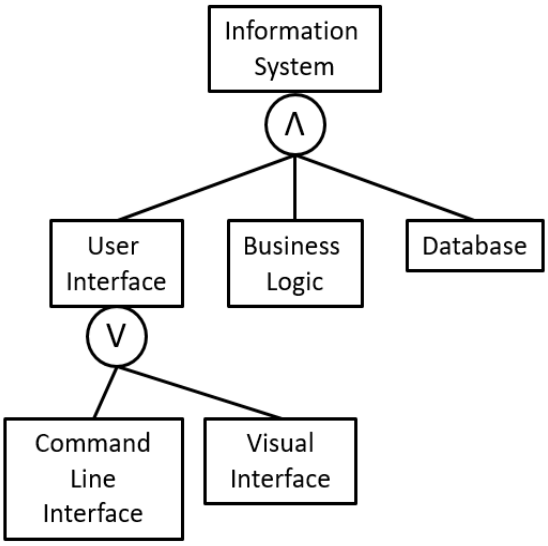

Fig. 5: FODA Syntax variation

Our syntax for representing concerns is based on *FODA* diagrams. In order to make some semantic extensions without looking too complicated however, a few modifications are made to the syntax to make it more extendible. As can be seen in Fig 5. logical operators are written in little circles that connect the children of the respective node. The roles of conjunction and disjunction are the same but now aggregation and other operators can be easily added.

The biggest difference of our fuzzy logic computation from the classical fuzzy logic approaches is the employment of non-truth-functional operators the details of which can be found in [16]. To summarize the idea behind this approach we can say that the *correlation* between the two statements are taken into consideration when computing any logical operator that joins these two statements. So in this approach the operands are not mere numbers, but the information about the statements themselves has to be preserved. This is mainly why an *Ontology Base* is required for correctly computing the concerns.

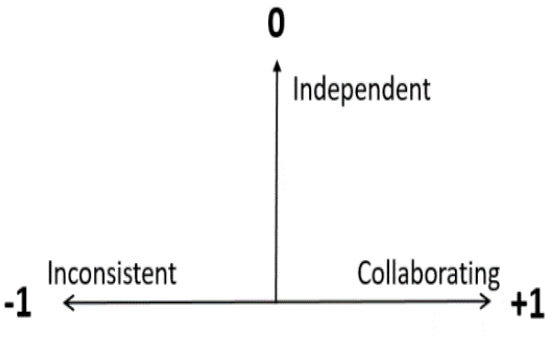

Fig. 6: Directions of interaction of statements

Fig 6. depicts the possible variation of the *correlation* between any given two concepts. The concepts can be *collaborating* in the sense that a high value in one concept can be a sign of a high value in the other (*correlation=1*). They can be *inconsistent* so that a high value in the first concept prevents a high value in the other (*correlation=-1*). They can also be independent from each other (*correlation=0*). As indicated in our former work [16], semantically *Gödel T-norm* is the suitable function for computing conjunction in the case of *collaboration* between the operands (this is also the most commonly used t-norm : the min function). In the case of *independent* operands *product t-norm* is the correct choice. If the operands are contradictory *Lukasievicz t-norm* should be preferred.

For the cases in between these formulas should be combined (at least the two on the sides should be individually combined with the *product t-norm*). We used the parametric general *t-norm* family called *Frank t-norms* for a smooth combination and also modified the formula just a little bit for allowing our correlation value (between -1 and 1) as a suitable parameter into the *Frank t-norm* formula. This makes a smooth value change between discrete correlation values possible. All *t-norms* also have corresponding *t-conorms* that can be computed directly from the *t-norm* formula. A corresponding *material implication* function can also be computed.

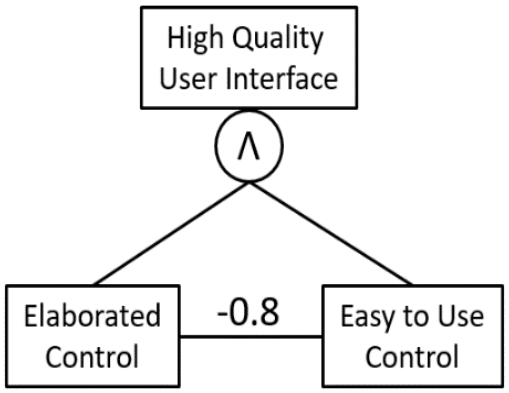

Fig. 7: Conceptual conjunction

In Fig 7. a piece of augmented tree that indicates a fuzzy conjunction can be seen. The conjunction defines a feature called "*High Quality User Interface*" by combining "Elaborated Control" with "*Easy to Use Control*". A line between the operands indicates the *correlation* between these two features and the value *-0.8* means that the two are almost *contradictory* (value is close to -1). In this case the requirement manager will use a formula that is very close to the *Lukasievicz t-norm*, with just a little touch of *product t-norm*.

Such design trade-offs are a natural sources of *negative correlation* but in fact they are the areas where the essence of good design comes from. When two forces counteract and create tension, creativity appears, not only in nature but also in human.

Another source of *negative correlation* is of course the multiplicity of the source of requirements. Basically there can be too many stakeholders in a project with conflicting interests. For example, in a hypothetical *school information system, the students, instructors, managers, the department, the faculty, and the student affairs bureau* all want to be able to monitor and manipulate everything, and conflicts may arise. One requirement originating from the *student affairs bureau* may ask for more security in accessing the data, while the *instructors* may demand more speed and

freedom when doing so. Without these problems we would not need designers and engineers.

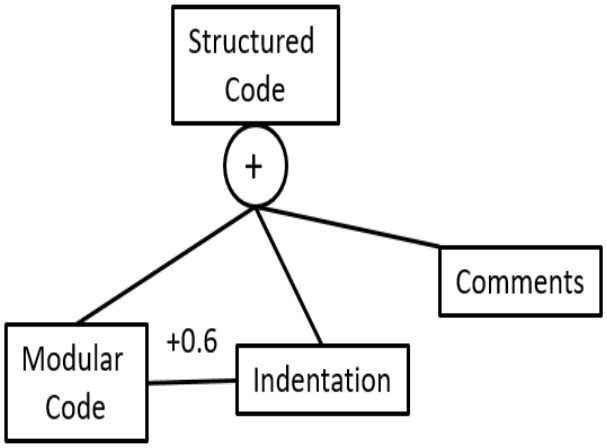

Fig. 8: Aggregation with overlap

A *positive correlation* can also interfere with the correct computation of concerns. In Fig 8. there is another piece of augmented tree that shows the definition of a concern called *Structured Code*. Instead of a conjunction or a disjunction, this feature has been defined by using the aggregation operator. *Aggregation* is used whenever there are multiple cooperating individuals that are performing different parts of the same task. A good example is the way pixels of a picture contribute to the overall color of the picture. The usual way of implementing aggregation is by an average (usually a weighted average) operator. When the operators of aggregation semantically overlap, their weight should be decreased accordingly.

In the example *Structured Code* was defined as a combination of *Modular Code, Indentation* and *Comments*. Between *Modular Code* and *Indentation* there has been a *positive correlation* indicated. This is because as the code is being modularized chunks of copy-pasted code are replaced by function calls, blocks get smaller and indentation problem is also partially solved. In this example Modular Code and Indentation will be assumed to be overlapping by 0.6 and their weights will be computed accordingly. There has been no correlation specified for *Comments* so zero correlation is assumed.

There can be cases when two requirements specified by two different stakeholders overlap completely (*correlation=1*). One requirement may be containing the other completely and in that case the smaller one has to be discarded. If the ontology does not inform about the containment relation the two values can be averaged separately and then enter the greater aggregation as one unit. If one requirement is just a redefinition of the same phenomenon, the concern should be corrected by deleting one of them.

An aggregation operator can be modified in other ways as may be seen in Fig 9.

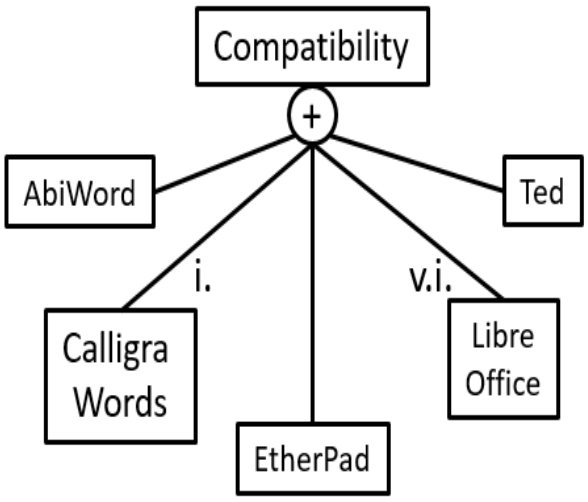

Fig. 9: Compatibility as an aggregation of features

Fig 9. represents a data compatibility requirement for a word processor. The new word processor that is being developed has to be compatible with various existing freely available word processors. The *concern* tree computes the overall degree of compatibility of the new product in terms of its degrees of compatibility with each of the existing word processors. These are all fuzzy values since the sets of supported files may intersect at various degrees.

None of the features on the tree are really indispensable but we cannot claim a *disjunctive combination* either. Disjunction indicates that *even one is enough*, although in this case the situation is rather like "*more the better*". However, the designer may still feel that one or two of these contributors are a little more important. This can be reflected on the tree by adding the labels *"i." (important)* and *"v.i." (very important)* on the links of the operands. In this case the requirement manager will still calculate the aggregation value but will make the necessary fuzzy touches to the weights of the corresponding operands just as much as the fuzzy terms "*important*" and "*very important*" inspire.

V. CONCLUSION

Requirement engineering is an extremely important part of software engineering and in large projects, automated support is a necessity. It is also a dynamic process that involves scalable behavior based on scalable judgments. The developers also have to deal with interrelated, redundant, incomplete or contradictory information.

As a remedy, in this work the logical design of a new fuzzy logic based requirement manager facility has been introduced by mentioning its most striking solutions. These are an original two dimensional fuzzy value representation and our new semantic-aware, non-truth-functional method of performing fuzzy operations. These were combined on an extended version of *FODA* diagrams that can express some more semantic features.

As a future work we are hoping to complete the implementation of our tool and observe some intelligent behavior from it.

REFERENCES

[1] A. Chakraborty, M. Baowaly and A. Arefin, "The Role of Requirement Engineering in Software Development Life Cycle", Journal of Emerging Trends in Computing and Information Sciences, 3(5), 2012.

[2] M. Burgin and J. Debnath, "Fuzzyness and Imprecision in Software Engineering", 2006 World Automation Congress, 24-26 July 2006.

[3] D. Lima, F. Freitas and G. Campos, "A Fuzzy Approach to Requirements Prioritization", Springer, 2011.

[4] D. Yadav, S. Chaturvedi and R. Mısra, "Early Software Defects Prediction Using Fuzzy Logic", International Journal of Performability Engineering 8(4), 2012, pp. 399-408.

[5] X. Huang, L. Capretz ve J. Ren, "A Neuro-Fuzzy Model for Software Cost Estimation", Proceedings of the Third International Conference On Quality Software, IEEE, 2003.

[6] M. Nisar, Y. Wang and M. Elahi, "Software Development Effort Estimation Using Fuzzy Logic - A Survey", Fifth International Conference on Fuzzy Systems and Knowledge Discovery, IEEE, 2008, pp. 421-427.

[7] S. Aljahdali, A. Sheta, "Predicting the Reliability of Software Systems Using Fuzzy Logic", 2011 Eighth International Conference on Information Technology: New Generations, 2011.

[8] S. Alzu'bi, B. Hawashin and M. ElBes, "A Novel Recommender System based on Apriori Algorithm for Requirements Engineering", 2018 Fifth International Conference on Social Networks Analysis, Management and Security (SNAMS), 2018, pp. 323-327.

[9] J. Bubenko, C. Rolland and P. Loucopoulos, "Facilitating Fuzzy to Formal Requirements Modelling", IEEE, 1994.

[10] Y. S. Yegorov, V. R. Milov, A. S. Kvasov, "Formalization of Software Requirements for Information Systems Using Fuzzy Logic", International Conference Information Technologies in Business and Industry 2018, IOP Publishing, 2018, pp. 1-5.

[11] M. Ramzan, M. ArfanJaffar M. AmjadIqbal, "Value Based Fuzzy Requirement Prioritization and its Evaluation Framework", 2009 Fourth International Conference on Innovative Computing, Information and Control, 2009, pp. 1464-1468.

[12] P. Ebraert, D. Soetens and D. Janssens, "Change-based FODA Diagrams Bridging the Gap Between Feature-oriented Design and Implementation", Conference: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011.

[13] M. Goncalves, R.Rodriguez, L.Tineo, "Formal Method to Implement Fuzzy Requirements", Dyna (Medellin, Colombia), 173(II), 2012, pp.15-24.

[14] X. Liu, "Fuzzy Requirements", IEEE Potentials, Institute of Electrical and Electronics Engineers (IEEE), Apr, 1998.

[15] M. Hsieh, Y. Hsu and C. Lin, "Risk Assessment in New Software Development Projects at The Frontend: A Fuzzy Logic Approach", Journal of Ambient Intelligence and Humanized Computing, 9(2), Springer, Apr, 2018.

[16] A. Egesoy, "Choosing Fuzzy Operators for Real-Life Engineering Applications", Turkish Journal of Fuzzy Systems, 8(2), 2017, pp. 73-89.

[17] N. D. Belnap, "A Useful Four-valued Logic", In: G. Epstein and J. M. Dunn (eds.), Modern Uses of Multiple-Valued Logic, Reidel, Dordrecht, 1977, pp. 7–37.

[18] P. Pohjalainen, "Feature Oriented Domain Analysis Expressions", Computer Science, 2008.

ABOUT THE AUTHORS

Ahmet Egesoy (PhD in Computer Engineering) is an instructor and Assistant Professor in Computer Engineering Department of Ege University Izmir, Turkey.
Research interests include object-oriented programming, design patterns, model-driven software development, artificial intelligence, programming languages, programming paradigms, philosophy of the language, semiotics and knowledge representation.

Aylin Güzel (M.Sc. in Computer Engineering) is a Ph.D. student in Computer Engineering Department of Ege University, Izmir, Turkey. Research interests include software engineering, object oriented programming, fuzzy logic and design patterns.