A FUZZY MODEL BASED ON SOFTWARE QUALITY METRICS WHICH ESTIMATES SOFTWARE MAINTAINABILITY

J. N. Gichuru[1], W. Mwangi[2] and M. Kanyaru[3]
[1,2]Institute of Conputer Science and information Technology, Jomo Kenyatta University of Agriculture and Technology, Nairobi, Kenya
[3]Software Systems Research Center, Bournemouth University, United Kingdom
E-mail: joangichuru@yahoo.com

Abstract
This study proposes a prediction model built on fuzzy logic technology to estimate the maintainability of a software product. This research is guided by two objectives: First is to establish the factors that determine software maintainability at source-code level and the metrics that capture these factors. Second is to establish a means of combining these metrics and weigh them against each other. The outcomes of these objectives are presented as well as a discussion of knowledge modeling using fuzzy logic. The development of this model is based on the fact that maintainability like other software quality facets can be described in terms of a hierarchy. This hierarchy consists of factors, attributes and metrics. The model captures factors that determine maintainability at source-code level as articulated by various attributes. Three metrics which quantify these attributes are then considered as input parameters to the model. These metrics are average cyclomatic complexity, average number of live variables and the average life span of variables. Fuzzy logic is then used to weigh the metrics against each other and combine them into one output value which is the estimated software maintainability. This work is a contribution to the on-going research aimed at establishing a means to quantify maintainability of software. It is also an improvement to the much criticized maintainability index (MI), the identified measure so far.

Keywords: Software maintainability, fuzzy logic, average cyclomatic complexity, average live variables, average variable span

1.0 Introduction

Software maintenance encompasses a broad range of activities including error corrections, enhancement of capabilities, deletion of obsolete capabilities and optimization (Lamb, 1988). Software maintainability is dependent on various factors ranging from personnel-skill, the environment of development, size and age of an application. In addition, program features at source-code level largely determine the maintainability of the resulting software. These features include the number of branches in a module, the degree to which a module calls other modules and the extent to which a module makes access to global variables (Sneed and Mercy, 1985).

Software maintenance occurs because software does not do what it was designed to do. Higher quality software is likely to require less maintenance. The need to improve the quality of software makes it necessary to have a way of quantitatively measuring quality and this is the realm of software quality metrics. Software quality metrics are a crucial tool when it comes to solving software maintenance problems. The metrics quantitatively measure aspects of a system which can be used as indications of the overall quality of a system (Elshoff, 1998).

The problem of maintaining software is widely acknowledged in industry and much has been written on how maintainability can be facilitated. However, you cannot control what you cannot measure and there is yet no universal measure of maintainability. Some proposals have indeed been presented, but the idea of measuring maintainability has inherent problems. Firstly, maintainability has mainly been described using informal descriptions e.g. the IEEE Standard Glossary of Software Engineering Terminology of 1990 defines maintainability as the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment (IEEE, 1990). Measuring maintainability as “effort or ease” introduces a problem since the commonly known unit for effort is “man/months or man/hours”. Hence, informal definitions do not in any way guide in how to estimate or measure maintainability. The known measure of software maintainability so far is maintainability Index (MI) which also has notable gaps. MI is a composite metric that considers weighted Halstead metrics (effort or volume), McCabe’s Cyclomatic Complexity, lines of code (LOC) and number of comments (Stephen, 2003). Practically though, the involvement of comments in the MI has posed as a difficulty in understanding maintainability. This is because comments, just like source code, will degrade over time as maintenance activities are performed unless specific actions are taken to keep them from becoming inaccurate. Again, comments are not executed at run-time but are meant for people. Only people therefore can tell if the comments in the code are helpful or not hence their inclusion in code does not necessarily mean that the code is more maintainable.

ISO (2000) quality model describes maintainability as the set of attributes that bear on the effort needed to make specified modifications on software. Following this description, maintainability may be understood as a function of directly measurable attributes. This approach is practical especially when there are defined metrics to measure those attributes. However, there may be difficulty in weighting the measures against each other and combining them in a function. There are thus several issues surrounding estimation of software maintainability.

This paper proposes a model that uses three source-code level metrics namely average number of life variables, average life span of live variables and average Cyclomatic complexity. These three metrics will form the inputs to the model which by applying fuzzy logic will compute an estimate of the expected maintainability. These metrics reflect source-code level features that determine maintainability such as branching in modules, nature of variable access and interdependence.

2.0 Materials and Methods

2.1 Approach

The model follows the approach that maintainability is a function of directly measurable attributes as described by ISO (2000). These attributes are features of source-code that determine maintainability of the resulting software such as nature of branching in modules, variable access and inter-dependence of modules. These features will be captured using three defined metrics namely; average Cyclomatic complexity, average number of live variables and average span of variables which will form inputs to the proposed fuzzy model. Fuzzy logic will be used to weight the measures against each other and to combine them into one function. The model will thus have a single output which is the estimated maintainability of the program being evaluated.

2.2 Development Platform

These computations will be done using MATLAB while the design of the model will be implemented using Fuzzy Toolbox, an add-on tool in MATLAB. MATLAB is a high performance language for technical computing which integrates computation and programming. It makes use of mathematical notations to express the problems and solutions being modeled. It has several add-on toolboxes which are suited for various specialized technology. Among them is the Fuzzy Logic Toolbox which is a collection of functions built on the MATLAB technical computing environment. It enhances creation and editing of fuzzy inference systems within the framework of MATLAB.

2.3 Fuzzy Logic

Fuzzy logic is an innovative technology that enables one to describe a desired system behaviour using every day spoken language. It is a natural continuous logic that is patterned after the approximate reasoning of human beings. Approximate reasoning refers to processes by which imprecise conclusions are inferred from imprecise premises (Nguyen and Walker, 1999). The context in which theory of approximate reasoning is used is manifested by domain knowledge containing uncertainty and imprecision. When the domain knowledge contains fuzzy concepts and is expressed in a natural language, then fuzzy logic becomes an appropriate theory for modeling (Canfora et al., 1998). Unlike traditional or Boolean logic, it's not restricted to the conventional binary computer values of zero and one but it allows for partial truths and multi-valued truths. Fuzzy logic recognizes the advantages of approximate reasoning whereby in most real world situations, a precise answer does not necessarily provide an optimal solution. It is particularly useful for problems that cannot be easily represented by mathematical modeling because data is either unavailable, incomplete or the process is too complex.

2.4 Factors Affecting Software Maintainability

In software engineering, maintainability is defined as the ease with which a software product can be modified. A software product is modified in order to correct defects, meet new requirements, make future maintenance easier, or cope with a changed environment. These tasks are commonly known as software maintenance activities (Sommerville, 2006). Several factors determine the ease with which maintenance activities can be done. These factors include program features at source-code level, age and size of a product, the development environment, personnel-skill among others. Personnel skill means the presence of or lack of a trained maintenance team. It is easier when modifications to a product are done by staffs who developed it or who have interacted with it while in use rather than new staff. Large products may be harder to modify than smaller applications even though it is not always the case. The age of a product also determines its ease of modification. Old products that have gone through several changes degrade and become less maintainable as compared to new products (Sommerville, 2006).

This study focuses on features of source-code that may determine maintainability. These features include dependency or coupling among modules, pattern of variable access and code complexity. Coupling is when one module modifies or relies on the internal workings of another module e.g. by accessing local data of

another module (Daly et al., 1996). This usually implies that changing the second module will lead to changing the dependent module. It is harder to perform maintenance activities on tightly coupled program code since a change in one module usually forces a ripple-effect of changes in other modules. Secondly, when a variable is declared as global it means that it can be accessed from all scopes and hence any code anywhere in the program can change the value of the variable at any time. The use of global variables makes it more difficult to isolate units of code for purposes of modification or even unit testing. The values of global variables can also be changed by any function that is called, and there is no easy way for the programmer to know that this will happen. Global variables therefore have unlimited potential for creating mutual dependencies which in turn increases complexity in the languages that implement them. The complexity of code also affects its maintainability since to a large extent it reflects difficulty in comprehending the code. Code complexity is articulated by the nature of control flow of modules, the flow of data and even the data structures used. Control flow is a general concept relating to the order in which various instructions of a program are executed (Coleman *et al*., 1994).

2.5 Metrics for the Study

Rombach has published the results of a carefully designed experiment that indicates that software complexity metrics can be used effectively to explain or predict the maintainability of software in a distributed computer system (Rombach, 1987). A set of three complexity metrics, each of which measures an aspect of structural complexity of a program code have been considered. These metrics are average number of live variables, average variable span (Peng and Dolores, 1994) and average cyclomatic complexity defined by McCabe (McCabe, 1976).

Live variable is a measure of data flow reflecting the number of variables whose values could change during the execution of a program. A variable is live at a particular statement if it is referenced by a certain number of statements before or after that statement (Peng and Dolores, 1994). The average number of live variables is therefore a ratio of the count of live variables and the count of executable statements. For example, in a section of code, the average number of live variables is the sum of the number of live variables for each line divided by the number of lines of code. The more the number of live variables, the more difficult it would be to develop and to maintain software.

Variable span is the number of source-code statements between successive references to a variable without consideration of comment lines (Peng & Dolores, 1994). For example, if a variable is referenced in lines 13, 18, 20, 21 and 23, the average span would be the sum of all spans divided by the count of spans, i.e., $(4+1+0+1)/4 = 1.5$. A large value for variable span reduces understandability and readability of code and a far back reference is likely to be missed out or forgotten.

Cyclomatic complexity also measures structural complexity and depicts the flow of control in a program. It considers decision points in a method or module which are implemented by use of conditional statements like if–else or while and logical operations such as AND, OR and NOT is defined as average of Cyclomatic complexities of all the modules (McCabe, 1976). Given any computer program, we can draw its control flow graph, G wherein each node corresponds to a block of sequential code and each arc corresponds to a branch or decision point in the program. The Cyclomatic complexity of such a graph can be computed by a simple formula from graph theory; $v(G) = e - n + 2$ where e is number of edges, and n is the number of nodes in the graph. Cyclomatic complexity has enormous impact on the testability and maintain-ability of code. If the Cyclomatic complexity value increases, there is an increasing number of decision points within the code and this means more effort to test the method.

2.6 Procedure

This section describes a step by step detailed design of the fuzzy model. It entails the following four steps:

2.6.1 Definition of Linguistic Variables

The model will consider 3 input variables namely: average Cyclomatic complexity, average number of live variables and average span of a variable. In the design of the model, these variables will be denoted as CycCom, LivVar and VarSpan respectively. Each of these variables will be described using three linguistic terms namely Low, Medium and High. The model has one output variable namely maintainability denoted as MANT which will be described using five linguistic terms to give more precision to the output obtained. These terms are namely Very-low, Low, Medium, High and Very-high.

2.6.2 Definition of Membership Functions

Standard membership functions will be used due to their simplicity and ease of interpretation.
These standard membership functions include Z-shaped, S-shaped and triangular-shaped membership functions named after the shape of their plots. (Canfora *et al*.,1998). The variables are fuzzified as shown in the figures below.

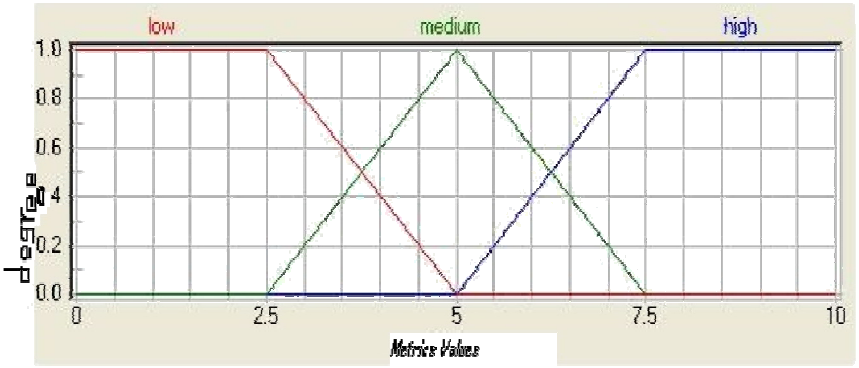

Figure 1: Fuzzification of cyclomatic complexity

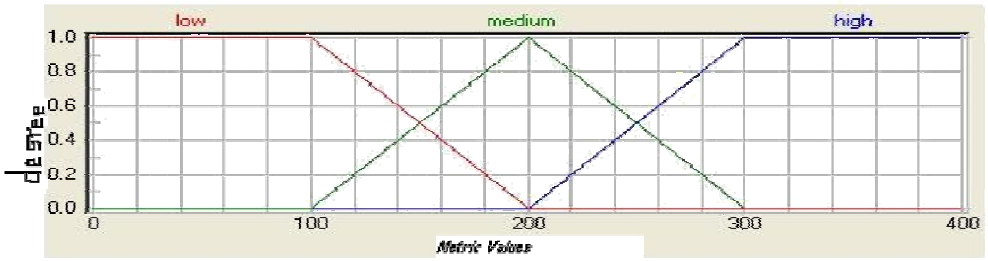

Figure 2: Fuzzification of variable span

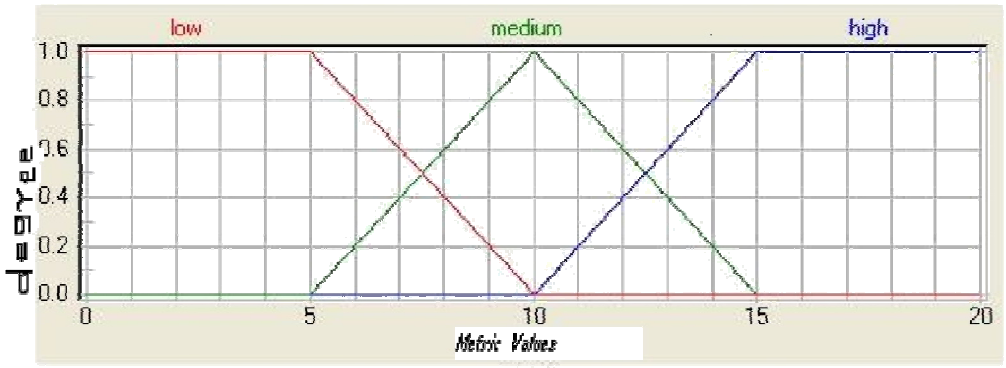

Figure 3: Fuzzification of count of live variables

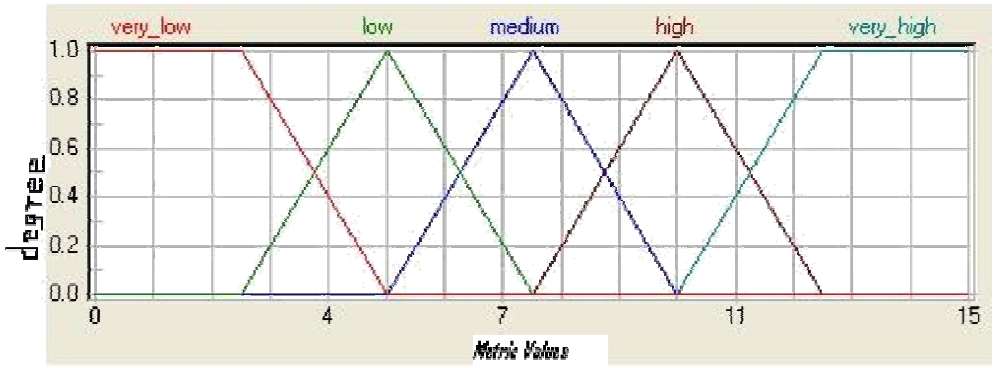

Figure 4: Fuzzification of maintainability

2.6.3 Creation of a Rule Base

A knowledge base or rule base consisting of 27 rules is created. This number is as a result of considering all the possible combinations of the three inputs i.e. 3^3 and a rule is assigned to each combination. The rules are expressed in verbose format where the number in the brackets represents the weight e.g.

If (Cyclomatic-Complexity is Low) and (Live-Variable is Low) and (Variable- Span is Low) then (Maintainability is very good) (1)

2.6.4 Selection of a Suitable Defuzzification Method

The choice of a defuzzification procedure is usually guided by considerations such as robustness or sensitivity to errors in its arguments. This research employs the centroid method sometimes called Center-of-Gravity method since its continuous on the space of membership functions hence tends to have good robust properties.

3.0 Results

Suppose we have the following inputs to the model as shown in the sample interface below: *CycCom*=2, *LivVar* =1 and VarSpan =130. When those inputs are fuzzified we find that *CycCom*=2 belongs to fuzzy set low with membership grade 1, *LivVar* =1 belongs to fuzzy set low with membership grade 1 and VarSpan =130

belongs to fuzzy set Medium with membership grade = 0.5 and medium with membership grade 0.5.With these inputs, rule 3 fires and an output value of 3 is obtained for MANT.

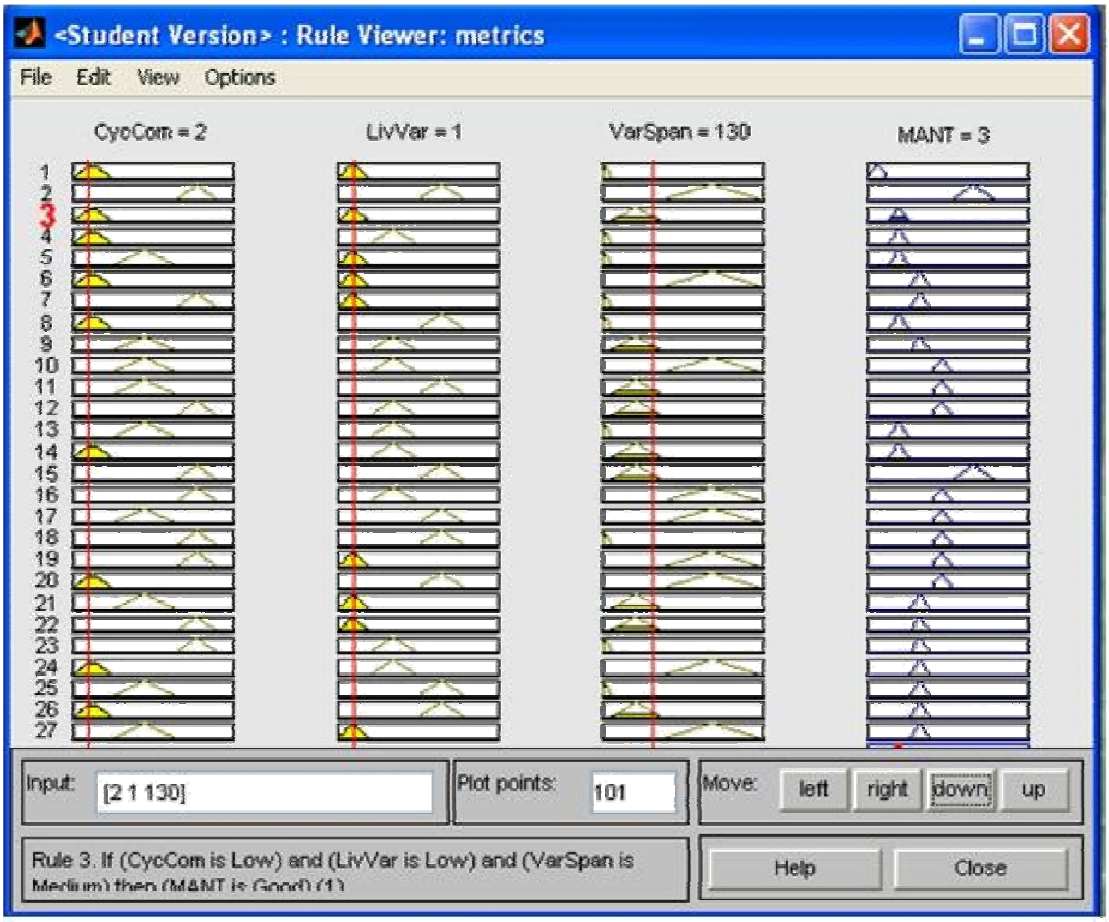

Figure 5: Model output computation

4.0 Discussion and Conclusions

To better understand these results, we look at maintainability as a software quality component that can be described in terms of a hierarchy. This hierarchy consists of factors, attributes and metrics. A factor is at the top-level and is an expression of the software status. Each factor is described by a set of attributes which can be measured by a set of metrics. In this view, the fuzzy model developed in this study is based on factors that affect software maintainability. These factors are articulated by measurable attributes that have been captured by the three metrics used. Since we understand how each metric has been defined and which attributes have been measured, then we increase our knowledge on the source-code process created and also the expected quality of that source code when placed into service. Parnas (2001) argued that the solution to the maintenance problems lies not in maintenance, but in development. Even during development, software is subject to constant change and revision. Use of prediction models is one among many software practices aimed at facilitating this and it is concluded from this study that fuzzy logic can be used to come up with a prediction model.

References

Aggarwal, K., Singh, Y., Chadra, P. and Puri, M. (2005). Measurement of software maintainability using fuzzy logic. *Journal of Computer Sciences*, **4**, pp 538–542.

Driankow, D., Hellendoorn H. and Reinfrank (1996). An Introduction to Fuzzy Control, 23–47. Springer-Verlag.

Daly, J., Brooks, A., Miller, J., Roper, M. and Wood, M. (1996). *Evaluating* inheritance depth on the maintainability of object-oriented software. *J. Empirical Software Engineering*, **1**, pp 109–132.

Elshoff, J. L. (1998). An investigation into the effects of counting method used on software science measures. *ACM SIGLPA Notices*, **13**,pp 30–45.

Kafura, D. and Reddy, G. (1987). The use of software quality metrics is software maintenance. In IEEE Trans. *Software Eng*., vol. **3**, 335–343

McCabe, T. J. (1976). A complexity measure. IEEE Trans. *Software Eng., SE*-**2**, pp 308-319.

Nguyen, H. T. and Walker, E.A. (1999). A First Course in Fuzzy Logic, 67–76. Chapman Hall, 2nd edn.

Rombach, H. D. (1987). A controlled experiment on the impact of software structure on maintainability. In IEEE Trans. *Software Eng.,* vol. **3**, pp 344–354.

Sneed, H. AND Mercy, A. (1985). Automated software quality assurance. IEEE Trans. Software Engineering, 909–916.

Sommerville, I. (2006). *Software Engineering*, 9–23. Pearson Education, 7th edn.

Stephen, H. K. (2003). Metrics and Models in Software Quality Engineeing, 234–257. Pearson Education Inc., 2nd edn.