JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2010**

Vol. 9, No. 3, May-June 2010

A Fuzzy Logic Approach to Measure Complexity of Generic Aspect-Oriented Systems

Rajesh Kumar, School of Mathematics & Computer Applications, Thapar University, Patiala, Punjab, India.
P.S. Grover, Guru Tegh Bahadur Institute of Technology, GGS Indraprastha University, Delhi, India.
Avadhesh Kumar, Galgotias College of Engineering & Technology, UP Technical University, Uttar Pradesh, Greater Noida, India.

Abstract

Aspect-oriented programming (AOP) is an emerging technique that provides a mechanism to clearly encapsulate and implement concerns that crosscut other modules. It is claimed that this technique improves code modularization and therefore reduces complexity of object-oriented programs (OOP). Most of the proposed complexity measurement frameworks for AOP are for AspectJ programming language. In this paper, a generalized framework for assessment of complexity of aspect-oriented (AO) systems, has been defined that takes into account three, the most well known families of available AOP languages, AspectJ, CaesarJ and Hyper/J. In order to automate complexity measurement, a tool has been developed using fuzzy logic, in which some set of rules have been defined as rule base. Using this tool, complexity of majority of AOP languages can be measured, which will further help in the measurement of external software qualities, such as maintainability, reusability, adaptability and understandability.

Keywords: aspect-oriented programming, complexity metrics, fuzzy logic.

1 INTRODUCTION

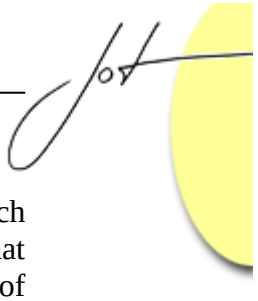
Now days, our society is becoming dependent on software that's why demand of quality software is increasing day by day. In the literature of software quality models, many researchers and practitioners have proposed their quality models, which are intended to evaluate external software qualities such as *maintainability, usability, efficiency, functionality, reliability, portability and reusability*. These external software quality characteristics could be measured with the help of software metrics. Metrics are designed on the basis of design structure of programming languages such as module-oriented

programming (MOP), object-oriented programming (OOP) and aspect-oriented programming (AOP) [1]. Design of metrics depends on internal quality characteristics such as *encapsulation, cohesion, coupling* and *complexity*. In turn, researchers and practitioners have proposed a large number of new metrics and assessment frameworks for quality design principles such as complexity [2, 3, 4, 5]. High complexity of any software system is an indication of low quality.

AOP languages aim to improve the ability of designers to modularize concerns that cannot be modularized using traditional module-oriented (MO) or object-oriented (OO) paradigms. Such concerns are scattered in multiple modules (classes) and are known as *crosscutting concerns* [6]. Examples of crosscutting concerns include *logging, tracing, caching, resource pooling* etc. The ability to modularize such concerns is expected to improve *comprehensibility, parallel development, reuse* and *ease of change* [7, 8], *reducing development costs*, increasing *dependability* and *adaptability*. Since AO is a new abstraction, the definition of complexity is required to redefine in the context of AOP.

Out of all available AOP languages, the most popular is AspectJ [9]. AspectJ is an extension of Java with several complementary mechanism, namely *join points (JPs), pointcut descriptors (PCDs), advice, introduction* and *aspect*. JPs represent well-defined points in a program's execution. Typical *join points* in AspectJ include method calls, access to class members, and the execution of exception handler blocks. A PCD is a language construct that picks out a set of *join points* based on defined criteria. The criteria can be explicit function names, or function names specified by wildcards. *Advice* is code that executes *before, after,* or *around* a join point. You define advice relative to a *pointcut*, saying something like "run this code before every method call I want to log". *Introduction* allows aspects to modify the static structure of a program. Using *introduction*, aspects can add new methods and variables to a class, declare that a class implements an interface, or convert checked to unchecked exceptions. Advice, pointcuts, ordinary data members and methods are grouped into class-like modules called *aspects*. Aspects are intended to support the modular representation of crosscutting concerns, although they admit other uses. Some existing AOP languages and frameworks provide a very similar composition model to the AspectJ one, such as Springs AOP framework [10] and JBoss AOP [11]. However, despite a good amount of work for measuring complexity, there is poor understanding of complexity in the context of AOP. Some researchers and practitioners have proposed complexity measurement frameworks and metrics for AOP [12, 13, 14]. But most of them are for AspectJ. They have defined complexity in context of AspectJ.

Another family of AOP languages is CaesarJ [15], which does not have aspects as a separate language abstraction. It supports additional concepts such as *virtual classes, mixin composition, aspectual polymorphism,* and *bindings*. IBM's Hyper/J [16], is also becoming popular as one of the AOP languages. When using Hyper/J, a developer provides three inputs: a *hyperspace* file that describes the Java class files, which can be manipulated by Hyper/J, a *concern mapping* file that describes, which pieces of the Java



source map to each dimension of concern, and a *hypermodule* file that describes which dimensions of concern should be integrated (i.e., which *hyperslices*) and how that integration should proceed. Individual *aspect* may be viewed as *hyperslice*, and the set of aspects together with the core classes as *hypermodule*. There is no central composition rule in AspectJ and CaesarJ. Instead, each aspect contains its part of the rule, specifying how that aspect is to be woven into the base classes. Despite a growing body of work dedicated to measure complexity in AO systems, there is no tool which could automate the assessment of complexity of generic AO systems. We have proposed new complexity metrics for generic AO systems and have developed a tool using fuzzy logic [17] to automate complexity measurement. There are three different inputs, which contribute in complexity of module/component in AO system. Ninety six fuzzy rules have been defined as rule base (knowledge base) to measure complexity of generic AO systems. This proposed framework has specifically targeted at the composition models supported by Java, AspectJ, CaesarJ and Hyper/J. This will help in: (i) defining new complexity metrics, which in turn, permits the analysis and comparison of Java, AspectJ, CaesarJ and Hyper/J implementations, (ii) integrating different existing measures, which examines the same concepts in different ways, and (iii) automate complexity measurement of majority of the AOP languages.

The paper is structured accordingly. Section 2 describes related work. Section 3 presents software complexity model for generic aspect-oriented programs. After defining complexity model in section 3, section 4 defines a fuzzy-logic approach to complexity metrics and model. Conclusions and future work are presented in section 5.

2 RELATED WORK

Xia et al. [18] described a new way of assigning complexity weight values to function point metric. They discussed the concepts of calibrating Function Points, whose aims are to estimate a more accurate software size that fits for specific software application, to reflect software industry trend, and to improve the cost estimation of software projects. In this paper, a Function Point calibration model called Neuro-Fuzzy Function Point Calibration Model (NFFPCM) that integrates the learning ability from neural network and the ability to capture human knowledge from fuzzy logic is proposed. The empirical validation using International Software Benchmarking Standards Group (ISBSG) data repository release 8 shows a 22% accuracy improvement of mean magnitude relative error (MMRE) in software effort estimation after calibration. This weight values assignment is for functions (operations) only, not for other members of the class and this framework is for object-oriented systems.

Sicilia et al. [12] talked about the main design and implementation issues aspect-oriented design (AOD)-based extensions on OJB database libraries using fuzzy logic. They specified that fuzziness can be considered as separate crosscutting concern in existing software, and in consequence, AOD techniques provide a convenient framework to implement fuzzy extensions to existing libraries.

Jana [13] measured code complexity in projects designed in AspectJ. They have defined and used entropy metrics for ordering of symbols to estimate the complexity of AspectJ based programs. The entropy metrics are useful in ranking different modules and symbols with regard to their complexity. They introduced weighted entropy values to accommodate the subjective perspective of an observer. Their approach provides multi valued space more suitable for prediction models. This framework is applicable only for AspectJ-like languages.

Norbert et al. [14] described a multi-paradigm metric which is extended for aspect-oriented programs. The metric can measure complexity of MO, OO and AO parts of programs implemented in AspectJ. This extended metric revealed that aspect-oriented does not necessarily reduce the complexity on its own- the gain highly dependent on the actual problems.

Kumar et al. [19] defined unified framework for cohesion measurement in AO systems. In their framework they considered Java, AspectJ and CaesarJ programming languages. Grover et al. [7] defined unified/generic AOP framework for changeability measurement using same terminologies and framework mentioned in [19]. Kumar et al. [20, 21] also defined new unified/generic framework for measuring coupling and complexity of AO systems. In this framework, they included Hyper/J, one of the popular AOP language, besides Java, AspectJ and CaesarJ. This paper is an extension of our earlier work [21].

3 SOFTWARE COMPLEXITY MODEL FOR GENERIC ASPECT-ORIENTED SYSTEM

In order to define generic complexity model which accounts Java, AspectJ, CaesarJ and Hyper/J as part of generic AO framework, first step will be to define (i) AO terminology and formalism for unambiguous and standardized representation and (ii) generic AO framework which can specify different aspects like inheritance, domain of measure, interaction type and so on. Since this work is an extension of our earlier work [21], in which we have already defined both the sections of (i) and (ii). So, there is no need to re-write the same thing here, but to understand terminologies and formulism used in this paper, one has to refer [21].

We have divided complexity of generic AO system in two categories: (i) Code Complexity and (ii) Interaction Complexity, which could be defined as:

Complexity of AO system:

$$CMPX_{AOS} = CMPX_{C(S)} + CMPX_{IC(S)}$$

Where, $CMPX_{AOS}$, $CMPX_{C(S)}$ and $CMPX_{IC(S)}$ are complexity of AO system, code complexity of Component Set and complexity of interactions between the Components respectively. Scope of this paper is to measure code complexity only. Interaction complexity could be measured in the similar way.

Code Complexity of Component Set:

$$CMPX_{C(s)} = \sum_{x=1}^{X} CMPX_{M(c_x)}$$

Where, x is the total number of components in the AO system and $CMPX_{M(c_x)}$ is the code complexity of a component c_x.

Code Complexity of a Component:

$$CMPX_{M(C)} = \alpha \times CMPX_{Att(c)} + \beta \times CMPX_{Op(c)} + \gamma \times CMPX_{Nested(c)}$$

Where, α, β and γ are the coefficients for $CMPX_{Att(c)}$, $CMPX_{Op(c)}$ and $CMPX_{Nested(c)}$ respectively and are dependent on the nature of components. $CMPX_{Att(c)}$, $CMPX_{Op(c)}$ and $CMPX_{Nested(c)}$ are attributes complexity, operations complexity and nested components complexity respectively in a component c.

Complexity of Attributes:

$$CMPX_{Att(c)} = \sum_{l=1}^{L} w_l \times Att_l(c)$$

Where, L is total number of attributes in a component c and w_l is the corresponding weight value of an attribute $Att_l(c)$.

Complexity of Operations:

$$CMPX_{Op(c)} = \sum_{m=1}^{M} w_m \times Op_m(c)$$

Where, M is total number of operations in a component c and w_m is the corresponding weight value of an operation $Op_m(c)$.

Complexity of Nested Components:

$$CMPX_{Nested(c)} = \sum_{n=1}^{N} w_n \times Nested_n(c)$$

Where, N is total number of nested components in a component c and w_n is the corresponding weight value of a nested component $Nested_n(c)$.

Now, we can represent Complexity of Component Set as:

$$CMPX_{C(s)} = \sum_{x=1}^{X} \left(\alpha \times \sum_{l=1}^{L} w_l \times Att_l(c_x) + \beta \times \sum_{m=1}^{M} w_m \times Op_m(c_x) + \gamma \times \sum_{n=1}^{N} w_n \times Nested_n(c_x) \right)$$

It is important to note that we can add above complexity metrics only when all metrics values are in same unit/scale. For example, if one metrics is in a/b format and ranges between 0 to 1, then others must be in same unit to add values. If scales are different then there is a need to normalize units so that all metrics are of same units. It is very difficult to assign numeric values to weight values w_l, w_m and w_n. For example w_m is numeric value of complexity weight value of an operation. Operation complexity depends on return

type, number of parameters and their parameter type. Return type and parameter type may be built in types, user defined types and component types. Due to different number of parameters, different parameter types and different return types, there will be hundreds of combinations of complexity weight values of operations. Similarly we can assign different numeric values to w_l and w_n which may be hundreds in numbers. Complexity of attributes, complexity of operations and complexity of nested components are different in nature and have different type of complexity value. $CMPX_{Att(c)}$ is due to data, $CMPX_{Op(c)}$ is due to data as well as business logic (set of instructions) implemented for the data and $CMPX_{Nested(c)}$ is contribution of both $CMPX_{Att(c)}$ and $CMPX_{Op(c)}$. So, these complexities cannot be added to get $CMPX_{M(C)}$ of a component. Solution of this problem is proposed in section 4.1 and 4.2.

4 A FUZZY-LOGIC APPROACH TO COMPLEXITY METRICS AND MODEL

The solution that is suggested here to overcome previously mentioned problems is to use fuzzy logic linguistic variables for the complexity metrics and model. Fuzzy logic is a mathematical tool for dealing with uncertainties and also it provides a technique to deal with imprecision and information granularity. Fuzzy logic is seen as a means of approximate reasoning .Our fuzzy model for integrating AO component complexity $CMPX_{M(C)}$ accounts the effect of complexity of attributes $CMPX_{Att(c)}$, complexity of operations $CMPX_{Op(c)}$ and complexity of nested components $CMPX_{Nested(c)}$. A block diagram for the fuzzy model is shown in Fig-I.

The fuzzy model consists of four modules. The fuzzification module is the first stage in working of any fuzzy model, which transforms crisp input(s) into fuzzy values. In the second stage, these values are processed in the fuzzy domain by interface engine based on production rules (knowledge base) supplied by the domain expert(s). During second stage, the fuzzy operators are applied. In third stage implication process is applied and then all outputs are aggregated. In fourth and final stage, the processed output is transformed from fuzzy domain to crisp domain by defuzzification module.

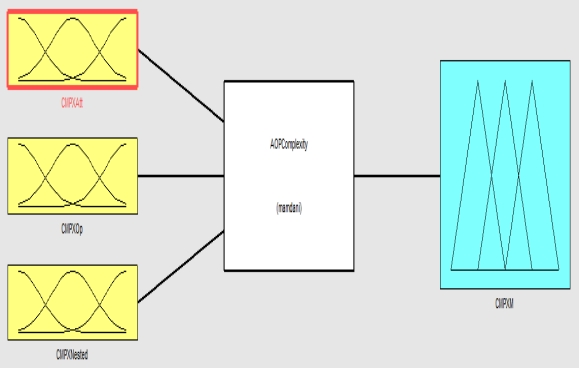


Fig-I: Fuzzy model for complexity measurement of a component.

4.1 Membership Functions for Input Parameters

In this paper, complexity of component ($CMPX_{M(C)}$) have been taken in the scale of 0 to 1 and member functions as NIL, Very Low (VL), Low (L), Medium (M), High (H) and Very High (VH). Because nested component is also a component, member function of nested component will be same as of a component i.e. NIL, VL, L, M, H and VH. Complexity of nested component ($CMPX_{Nested(c)}$) can be evaluated recursively with terminal condition as the component is without nested component i.e. $CMPX_{Nested(c)}$ as NIL for that component. For simplification, $CMPX_{Att(c)}$ and $CMPX_{Op(c)}$ values also we have taken in the range of 0 to 1. For $CMPX_{Att(c)}$ and $CMPX_{Op(c)}$, member functions have been considered as NIL, L, M and H. Now the question is, how to decide whether $CMPX_{Att(c)}$ is NIL, L, M or H? $CMPX_{Att(c)}$ value depends on number of attributes and their data types. In the section of terminology and formulism [21], there are three broad categories of types as built in types (BT), user defined types (UDT) and component types (CT). If complexity weight value assigned to individual attribute is taken in the range of 0 to 1, then we can assign weight values to the three types as given in Table-I.

Attribute Types	Complexity weight value
BT	0.33
UDT	0.66
CT	1.00

Table-I: complexity weight values to individual attribute

In order to evaluate complexity of all the attributes of a component, i.e. $CMPX_{Att(c)}$, we can apply formula given in the section 3. For example, in a component, there are 3 attributes of type BT, 4 attributes of type UDT and 2 attributes of type CT, then $CMPX_{Att(c)}$ will be 5.63 (3*0.33+4*0.66+2*1.00). For a component without attribute, $CMPX_{Att(c)}$ will be NIL. In order to decide whether this value (5.63) of complexity of attributes falls in category of

L, M or H, we performed experiment on benchmark projects [22, 23] and found that the maximum value of $CMPX_{Att(c)}$ was 10.61 (9*0.33+4*0.66+5*1), and in many other components this value was reaching 10. In other projects, this value may vary but if there is major difference, then proper decomposition of components at design time have not been done. For simplification, this maximum value of $CMPX_{Att(c)}$ has been normalized in the scale of 0 to 1. Range of member functions NIL, L, M and H of variable $CMPX_{Att(c)}$ are 0.00-0.00, 0.00-0.36, 0.33-0.68 and 0.65-1.00 respectively. There is overlapping in member function values for better results in fuzzy system. This is also shown in Fig-II.

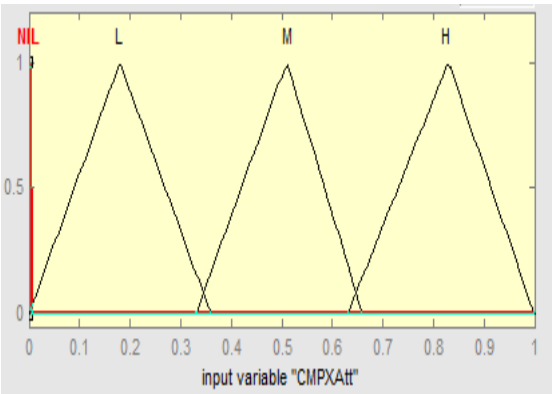


Fig-II

Complexity of operation depends on its return type, number of input parameters and parameter types, because business logic written in an operation depends on these. Complexity of parameters, we can evaluate using same methodology as applied on attributes. We can define whether an operation is simple type, medium type, complex type or very complex type; by input parameter complexity and return type. By the experience and expertise opinion of the field, we have defined the different types of operations, which are listed in Table-II.

Return Type	**Input Parameters Complexity**	**Operation Type**
Void	NIL	Simple
Void	L	Simple
Void	M	Simple
Void	H	Medium
BT	NIL	Simple
BT	L	Simple
BT	M	Medium
BT	H	Complex
UDT	NIL	Simple
UDT	L	Medium
UDT	M	Medium
UDT	H	Complex

CT	NIL	Medium
CT	L	Medium
CT	M	Complex
CT	H	Very Complex

Table-II: type of operations.

After categorizing operation types, we can assign complexity weight values to individual operations in the scale of 0 to 1 similar as in case of attributes, these are given in Table-III.

Operation Types	**Complexity Weight Value to Individual Operation**
Simple Type	0.25
Medium Type	0.50
Complex Type	0.75
Very Complex Type	1.00

Table-III: complexity weight values of operations.

We can now measure $CMPX_{Op(c)}$ using formula given in section 3 and by using weight values from Table-III. For example, a component is having 3 simple type operations, 2 medium type operations, 3 complex type operations and 1 very complex type operations, then $CMPX_{Op(c)}$ for this component will be 5.0 (3*0.25+2*0.5+3*0.75+1*1.0). Here, the same question arises as in case of $CMPX_{Att(c)}$, whether this value of $CMPX_{Op(c)}$ is L, M or H. For the solution of this, we again performed experiment on same set of projects [22, 23] and found that maximum value of $CMPX_{Op(c)}$ was 6.5 (6*0.25+4*0.50+2*0.75+1*1.00). We also found that for many components $CMPX_{Op(c)}$ value is in between 5 and 6. This maximum value of $CMPX_{Op(c)}$ may vary from project to project, but if there is major variation, then it may be because of proper decomposition of components have not taken place. Range of member functions NIL, L, M and H of $CMPX_{Op(c)}$ variable have been considered as 0.0-0.0, 0.0-0.36, 0.33-0.69 and 0.66-1.00 respectively, which are also shown in Fig-III.

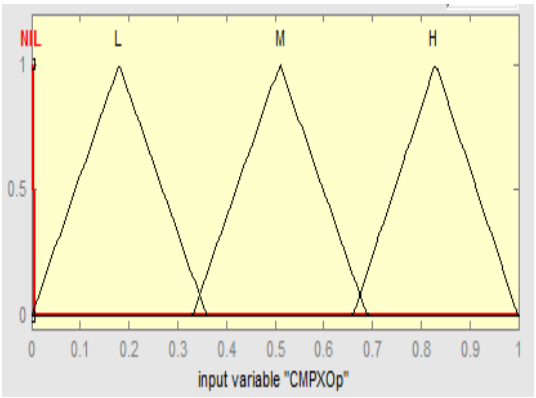


Fig-III

Range of member functions NIL, VL, L, M, H and VH of input variable $CMPX_{Nested(c)}$ and output variable $CMPX_{M(C)}$ have been considered as 0.0-0.0, 0.0-0.23, 0.2-0.43, 0.4-0.63, 0.6-0.83 and 0.8-1.00 respectively and for $CMPX_{Nested(c)}$ it is shown in Fig-IV, which will be similar for $CMPX_{M(C)}$.

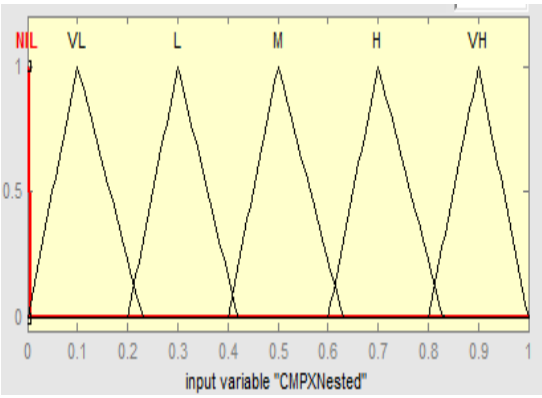


Fig-IV

4.2 Fuzzy Rules for the Proposed Model

In order to measure complexity of a component ($CMPX_{M(C)}$), which is the main objective of our model, there three members $CMPX_{Att(c)}$, $CMPX_{Op(c)}$ and $CMPX_{Nested(c)}$ contributing in the complexity of any component. Attributes, operations and nested components are different in nature and have different type of contribution in the complexity of a component, so we cannot simply add these values to get complexity of a component. As a solution of this problem, we have used fuzzy logic and have designed 96 fuzzy rules (4 member functions of $CMPX_{Att(c)}$ *4 member functions of $CMPX_{Op(c)}$ *6 member functions of $CMPX_{Nested(c)}$). Here, mamdani method for defining fuzzy rules is used, which is used for nonlinear equations. These rules are designed on the basis of experience and expertise knowledge of the field that's why these are also known as knowledge base. For sample, some of the rules are listed in Table-IV. First column labeled Rule# represent rule number, second column is for input linguistic variables, $CMPX_{Att(c)}$, $CMPX_{Op(c)}$ and $CMPX_{Nested(c)}$ and third column is for output linguistic variable $CMPX_{M(C)}$.

Rule#	**Input Variables**			**Output Variable**
	$CMPX_{Att(c)}$	$CMPX_{Op(c)}$	$CMPX_{Nested(c)}$	$CMPX_{Op(c)}$
1	NIL	NIL	NIL	NIL
12	NIL	NIL	VH	VH
25	L	NIL	NIL	VL
37	L	M	NIL	M
45	L	H	L	H

57	M	L	L	M
65	M	M	H	VH
93	H	H	NIL	H

Table-IV: Some Sample Rules of the Complexity Fuzzy Model

As an example, if $CMPX_{Att(c)}$ =0.573 (M), $CMPX_{Op(c)}$ =0.596 (M) and $CMPX_{Nested(c)}$ =0.5 (M) are input values then $CMPX_{Op(c)}$ value is resulting as 0.692, which is high for output variable $CMPX_{Op(c)}$. It is also shown in Fig-V as rule viewer.

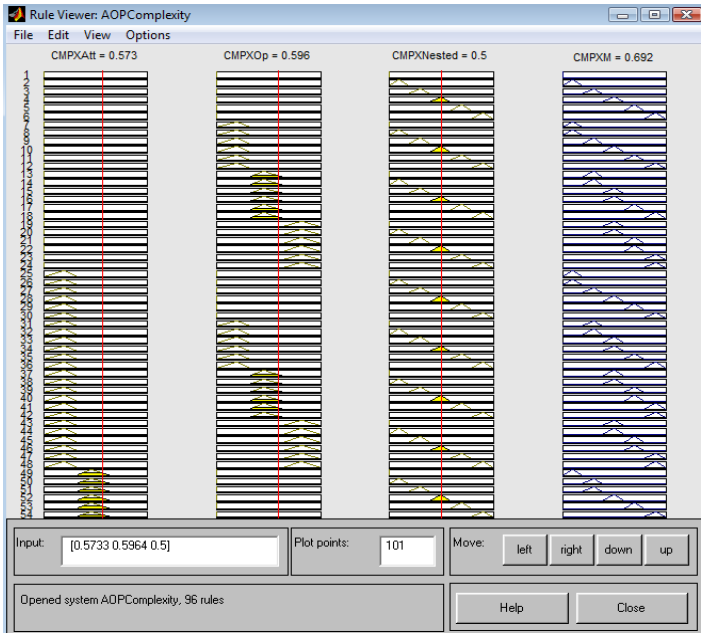


Fig-V:

Three dimensional surface view of this rule base is given in Fig-VI.

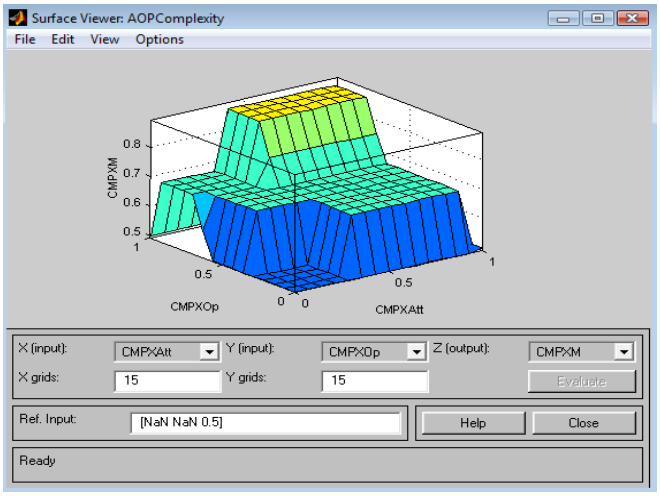


Fig-VI

Using proposed methodology and model, complexity of all the components of the software system can be measured and we can evaluate average of these complexity values. The average complexity value will be between 0 and 1and will be in any of the ranges, VL, L, M, H and VH. With the help of this value, we can specify complexity level of AO system.

5 CONCLUSION AND FUTURE WORK

In this paper, we have used fuzzy logic for defining software complexity metrics as linguistic variables and for the modeling process has been outlined. The motivation has been difficulties faced to get total complexity of a component in generic aspect-oriented system, because members, which contribute in the complexity of a component, are different in nature and have different type of complexity value. Using common terminology, formalism and generic/unified framework defined for Java, AspectJ, CaesarJ and Hyper/J, new complexity metrics have been defined. These metrics are defined for measuring code complexity and interaction complexity of AO system. In this paper, only code complexity has been evaluated. A fuzzy model has been defined to measure code complexity of a component. Average complexity of all the components available in the AO software system will be indicator to the complexity level of the system. Using this model, complexity of software developed in most of the AO languages including Java (OOP) can be measured, which further may be used as an indicator to external software quality such as *maintainability*, *reusability*, *adaptability* and *understandability*.

In future work, we have planned to measure interaction complexity of generic AO system using fuzzy logic approach and developing a model to get total complexity. In this paper, process of getting number of attributes, attribute types, number of operations, prototype of operations, numbers of nested components etc., is a manual process. This could be automated by developing a tool written in any programming language. We are in the process of developing this tool in Java, so that whole system of measuring complexity could be fully automated.

REFERENCES

[1] Avadhesh Kumar, Rajesh Kumar, P.S. Grover, "*A Comparative Study of Aspect-Oriented Methodology with Module-Oriented and Object-Oriented Methodologies*", ICFAI Journal of Information Technology, Vol. 2, No. 4, pp.7-15, December 2006.

[2] H.W. Schmidt and W. Zimmermann, *"A Complexity Calculus for Object-Oriented Programs"*, Journal of Object-Oriented Systems, Volume-1, Issue-2, pp. 117--147, 1994.

[3] Weyuker, E. J., "*Evaluating Software Complexity Measures*", IEEE Transactions on Software Engineering, Volume- 14,Issue- 9, pp: 1357-1365, Sep. 1988.

[4] Ognjen Prnjat, Lionel Sacks, "*Complexity Measurements of the Inter-Domain Management System Design*", Ninth IEEE International Conference on Networks (ICON'01), pp:2, 2001.

[5] Wilkie, F., "*Tool Support for Measuring Complexity in Heterogeneous Object-Oriented Software*", In Proceedings of the international Conference on Software Maintenance (ICSM'02), Washington, DC, October 03 - 06, 2002, pp: 152, IEEE Computer Society.

[6] V. C. Garcia, E. K. Piveta, D. Lucrédio, A. Álvaro, E. S. Almeida, L.C. Zancanella, & A.F. Prado, "*Manipulating crosscutting concerns*" , Proc. 4th Latin American Conf. on Patterns Languages of Programming (SugarLoafPLoP), Porto das Dunas, CE, Brazil, 2004.

[7] Avadhesh Kumar, Rajesh Kumar, P.S. Grover, *"An Evaluation of Maintainability of Aspect-Oriented Systems: a Practical Approach"*, International Journal of Computer Science and Security, Volume -1, Issue-2, pp. 1-9, Aug 2007.

[8] P.S. Grover, Rajesh Kumar, Avadhesh Kumar, "*Measuring Changeability for Generic Aspect-Oriented Systems*", ACM SIGSOFT Software Engineering Notes, Volume 33, Issue 6, pp-1-5, November 2008.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. "*An Overview of AspectJ*". In Proceedings of the 15th European Conference on Object-Oriented Programming, pp. 327–355, Springer, 2001.

[10] R. Johnson. Introducing the Spring framework., 2003. http://www.theserverside.com/tt/articles/article.tss? l=SpringFramework.

[11] J. Inc. JBoss AOP Beta3, 2004. http://www.jboss.org.

[12] Miguel-Ángel Sicilia, Elena García-Barriocana, "Extending Object Database Interfaces with Fuzziness Through Aspect-Oriented Design", ACM SIGMOD Record, Volume 35, Issue 2, pp: 4 – 9, June 2006.

[13] Jana Dospisil, *"Measuring Code Complexity in Projects Designed with AspectJ"*, Informing Science InSITE-"Where Parallels Intersects", pp: 185-197, June 2003.

[14] Norbert Pataki, Adam Sipos, Zoltan Porkolab, *"Measuring the Complexity of Aspect-Oriented Programs with Multiparadigm Metric"*, ECOOP 2006 Doctoral Symposium and PhD Students Workshop. http://www.ecoop.org/phdoos/ecoop2006ds/ws/pataki.pdf

[15] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. "*Overview of CaesarJ*", Transactions on AOSD I, LNCS, 3880: pp.135 – 173, 2006.

[16] Barry R. Pekilis, "*Multi- Dimensional Separation of Concerns and IBM Hyper/J*", Technical Research Report, January 22, 2002.

[17] Zadeh, L. A., Fuzzy Logic, Neural Networks, and Soft Computing, *Communications of the ACM,* Volume-37, Issue-3, pp: 77-84, Mar. 1994.

[18] Wei Xia, Luiz Fernando Capretz, Danny Ho and Faheem Ahmed, *"A new calibration for Function Point complexity weights"*, Information and Software Technology, Volume 50, Issue 7-8, pp: 670-683, June 2008.

[19] Avadhesh Kumar, Rajesh Kumar, P.S. Grover, *"Towards a Unified Framework for Cohesion Measurement in Aspect-Oriented Systems"* , 19th Australian Software Engineering Con-ference , 2008 (ASWEC 2008) Perth, Western Australia, pp.57-65, March 26-28, 2008 , **IEEE Computer Society.**

[20] Avadhesh Kumar, P.S. Grover, Rajesh Kumar, *"Generalized Coupling Measure for Aspect-Oriented Systems"*, **ACM** SIGSOFT Software Engineering Notes, Volume 34, Issue 3, pp:1-6 May- 2009.

[21] Avadhesh Kumar, Rajesh Kumar, P.S. Grover, *"Towards a Unified Framework for Complexity Measurement in Aspect-Oriented Systems"* , 2008 International Conference on Computer Science & Software Engineering (CSSE 2008), Wuhan, China, pp:98-103, Dec-12-14, 2008, **IEEE Computer Society.**

[22] http://caesarj.org/index.php/Caesar/Tutorial.

[23] Ramnivas Laddad, "AspectJ in Action: Practical Aspect-Oriented Programming", Manning Publications, 2003.

About the authors

Rajesh Kumar is presently Associate Professor and Head, Computer Centre at Thapar University, Patiala, Punjab, India. He received his Master and Doctorate degrees from Indian Institute of Technology (IIT), Roorkee. His area of research is Software Engineering focusing on Aspect-Oriented Programming, Component Based Software, Metrics and Software Quality. He has published more than 50 research papers in international and national journals of repute. He can be reached by e-mail at: rakumar@thapar.edu

P. S. Grover is presently Director General at Guru Tegh Bahadur Institute of Technology, GGS Indraprastha University, Delhi, India. Formerly he was Dean & Head of Computer Science Department, Delhi University, Delhi, India. He received his master's degree and doctorate from Delhi University, Delhi, India. He is widely travelled and delivere invited talks/key note addresses at many National/International Conferences/Seminars and Workshops. His current resrarch interests are: Component-based and Aspect-orinted Software Engineering and Autonomic Embedded Systems. He is on the Editorial Board of Four International Journals. Prof. Grover has written 9 books and many of his articles have appeared in several books published by IEEE of USA. He has published more than 100 research papers in international and national journals and conferences including published by IEEE, ACM and Springer. Dr. Grover is a member od IEEE Computer Society. He can be reached by e-mail at: groverps@hotmail.com

Avadhesh Kumar is presenty Associate Professor & Head, department of IT at Galgotias College of Engineering & Technology, UP Technical University, Uttar Pradesh, Greater Noida, India. He obtained his B.Tech. in Computer Science & Engineering from H.B.T.I. Kanpur, U.P., India and M.Tech. in IT from Punjabi University, Patiala, Punjab, India. He has submitted his Ph.D. thesis at Thapar University, Patiala, Punjab, India. His area of research is Software Engineering focusing on Aspect-Oriented Programming, Metrics, Software Quality and Component Based Systems. He is a member of IEEE, ACM & CSI. He has published more than 20 research papers in reputed international and national journals, and conferences including published by IEEE and ACM. He can be reached by e-mail at: kumar.avadh@gmail.com