FINDING FAULTS OF SOFTWARE METRICS FOR AN ARTIFICIAL IMMUNE RECOGNITION SYSTEM

S.Chitra [1] , K.Thiagarajan [2], Dr.M.Rajaram [3]

Abstract :

Software is an integral part of many critical and non-critical applications, and virtually any industry is dependent on computers for their basic functioning. Techniques to measure and ensure reliability of hardware have seen rapid advances, leaving Software as the bottleneck in achieving overall system reliability. Hence there rises a situation for the developers to develop high quality software. Having that in mind it is necessary to provide the reliability of the software to the developers before it is shipped. This can be achieved based on immune system. The immune system that is otherwise known as 'second brain' for its abilities to recognize new intruders and remember past occurrences. Simulating the immune system or translating immune system mechanisms into software learning is an interesting topic on its own. This will produce high quality, reliable results over a wide variety of problems compared to a range of other approaches, without the need of expert fine-tuning.

Index Terms - Artificial Recognition Balls, Artificial Immune Recognition System, Operating systems, software reliability growth models, Software Failure Rate.

Introduction:

The natural immune system is a powerful and robust information processing system that demonstrates several distinguishing features, such as distributed control, parallel processing, and adaptation/learning via experiences. Artificial Immune Systems (AIS) are emerging machine learning algorithms, which embody some of the principles of the natural immune system for tackling complex engineering problems [18]. The Artificial Immune Recognition System (AIRS), is a new supervised learning AIS. It has shown significant success in dealing with demanding classification tasks [19].

Software quality management is an important aspect of software project development. The Capability Maturity Model (CMM) is the *de facto* standard for rating how effective an organization's software development process is [20]. This model defines five levels of software process maturity: initial, repeatable, defined, managed, and optimization. The initial level presents the organizations with no project management system. The managed level describes the organizations, which collect information of software quality and development process, and use that information for process improvement. Finally, the optimization level describes those organizations that continually measure and improve their development process, while simultaneously explores the process innovations.

The software quality management is also an ongoing comparison of the actual quality of a product with its expected quality. Software metrics are the key tools in the software quality management, since they are essential indicators of software quality, such as, reliability, maintenance effort, and development cost. Many researchers have analyzed the connections between software metrics and code quality [21, 22]. The methods they use fall into the following main categories : association analysis, clustering analysis, classification and regression analysis .

In this paper, we propose an AIRS for the software quality classification. We also compare this method with other well-known classification techniques. In addition, we investigate the employment of the Gain Ratio (GR) for selecting relevant software metrics in order to improve the performance of the AIRS-based classifiers.

The remainder of this paper is organized as follows. Section 2 briefly introduces the software metrics and MDP benchmark dataset. Section 3 presents our AIRS based software quality classification method. Section 4 describes two baseline classification algorithms for comparison. Section 5 discusses the metrics selection with the Gain Ratio. Simulation results are demonstrated in Section 6. Finally, some remarks and conclusions are drawn in Section 7.

Software Metrics:

In this paper, we investigate totally 38 software metrics. Simple counting metrics, such as the number of lines of source codes or Halstead's number of operators and operands, describe how many "things" there are in a program. However, more complex metrics, e.g., McCabe's cyclomatic complexity or Bandwidth, attempt to describe the "complexity" of a program by measuring the number of decisions in a module or the average level of nesting in the module. These metrics are used in the NASA Metrics Data Program (MDP) benchmark dataset MW1 [23]. Refer to Table 1 for a more detailed description of the metrics. There are 403 modules in this dataset.

Our goal is to develop a prediction model of software quality, in which the number of defects associated with a module is projected on the basis of the values of the 37 software metrics characterizing a software module. We cast this problem in the setting of classification, and in each module, the explanatory variables are the first 37 software metrics, and the prediction variable is the defects. Software modules with no defects are in the class of *fault-none*, while those with more than one defect are in the class of *fault-prone*. Table 1. Description of NASA MDP MW1 project dataset with characterization of software metrics [24].

Software Metrics	Descriptions
LOC_BLANK	Number of blank lines in a module.
BRANCH_COUNT	Branch count metrics.
CALL_PAIRS	Number of calls to other functions in a module.
LOC_CODE_AND_COMMENT	Number of lines containing both code and comment in a module.
LOC_COMMENTS	Number of lines of comments in a module.
CONDITION_COUNT	Number of conditionals in a given module.
CYCLOMATIC_COMPLEXITY	Cyclomatic complexity of a module.
CYCLOMATIC_DENSITY	Ratio of the module's cyclomatic complexity to its length in NCSLOC. It factors out the size component of complexity, and normalizes the complexity and maintenance difficulty of a module.
DECISION_COUNT	Number of decision points in a given module.
DECISION_DENSITY	Calculated as: Cond. / Decision.
DESIGN_COMPLEXITY	Design complexity of a module.
DESIGN_DENSITY	Calculated as: iv(G)/v(G).
EDGE_COUNT	Number of edges found in a given module, which represents the transfer of control from one module to another. (Edges are a base metric, which is used to calculate involved complexity metric).
ESSENTIAL_COMPLEXITY	Essential complexity of a module.
ESSENTIAL_DENSITY	Calculated as: (ev(G)-1)/(v(G)1).
LOC_EXECUTABLE	Number of lines of executable code for a module (not blank or comment).
PARAMETER_COUNT	Number of parameters to a given module.
HALSTEAD_CONTENT	Halstead length content of a module.
HALSTEAD_DIFFICULTY	Halstead difficulty metric of a module.
HALSTEAD_EFFORT	Halstead effort metric of a module.
HALSTEAD_ERROR_EST	Halstead error estimate metric of a module.
HALSTEAD_LENGTH	Halstead length metric of a module.
HALSTEAD_LEVEL	Halstead level metric of a module.
HALSTEAD_PROG_TIME	Halstead programming time metric of a module.
HALSTEAD_VOLUME	Halstead volume metric of a module.
MAINTENANCE_SEVERITY	Calculated as: ev(G)/v(G).
MODIFIED_CONDITION_COUNT	
MULTIPLE_CONDITION_COUNT	Number of multiple conditions in a module.
NODE_COUNT	Number of nodes found in a given module. (Nodes are a base metric, which are used to calculate involved complexity metrics).
NORMALIZED_CYLOMATIC_COMPLEXITY	
NUM_OPERANDS	Number of operands in a module.
NUM_OPERATORS	Number of operators in a module.
NUM_UNIQUE_OPERANDS	Number of unique operands in a module.
NUM_UNIQUE_OPERATORS	Number of unique operators in a module.
NUMBER_OF_LINES	Number of lines in a module. Pure and simple count from open bracket to close bracket. It includes every line in between, regardless of character content.
PERCENT_COMMENTS	Calculated as: ((CLOC+C&SLOC)/(SLOC+CLOC+C&SLOC))*100.
LOC_TOTAL	Total number of lines for a given module.
ERROR_COUNT	Number of defects associated with a module.

Definition of AIRS:

The Artificial Immune Recognition System is a new method for data mining [25,26] People used to believe that "software never breaks". Intuitively, unlike mechanical parts such as bolts, levers, or electronic parts such as transistors, capacitor, software will stay "as is" unless there are problems in hardware that changes the storage content or data path. Software does not age, rust, wear-out, deform or crack. There is no environmental constraint for software to operate as long as the hardware processor it runs on can operate. Furthermore, software has no shape, color, material, mass. It can not be seen or touched, but it has a physical existence and is crucial to system functionality.

The presence of known vulnerabilities can represent an extremely high risk for some organizations such as banks, investment & brokerage houses, and web-based merchants. The software developers, and users need to be able to assess the risk posed by the vulnerabilities, and must invest in effective counter-measures. The risk increases with the delay in developing, and releasing a patch [1,2] A developer needs to allocate sufficient resources for continuous vulnerability testing, and patch development to stay ahead of the hackers. The users need to invest in data safeguard mechanisms, intrusion detection, and damage control. This investment must be proportional to the level of risk involved. Software reliability growth models [3,4] have been used for characterizing the defect-finding process for ordinary defects. Such models are used to assess the test resources needed to achieve the desired reliability level by the target date, and are needed for evaluating the reliability level achieved.

They can also be used to estimate the number of residual defects that are likely to be present. There is a need to develop similar models for quantitative characterization of the security aspects of the software. There are two separate processes to be considered: the first is the vulnerability discovery process, while the second is the exploitation of the individual vulnerabilities discovered. In this paper, we examine modeling the first process. An evaluation of the overall risk should involve a joint consideration of both processes. Obviously, vulnerability needs to be discovered before it can be exploited. Those who attempt to exploit vulnerabilities can often be amateurs because they can use the hacking scripts available on the Internet, which are developed after vulnerability has been reported. On the other hand, those who discover new vulnerabilities must have significant technical expertise, because the vulnerabilities often arise as the result of complex interactions of rarely occurring state combinations in the software.

According to ANSI, Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment. Although Software Reliability is defined as a probabilistic function, and comes with the notion of time, we must note that, different from traditional Hardware Reliability, Software Reliability is not a direct function of time. Electronic and mechanical parts may become "old" and wear out with time and usage, but software will not rust or wear-out during its life cycle. Software will not change over time unless intentionally changed or upgraded.

Software Reliability is an important to attribute of software quality, together with functionality, usability, performance, serviceability, capability, install ability, maintainability, and documentation. Software Reliability is hard to achieve, because the complexity of software tends to be high. While any system with a high degree of complexity, including software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the rapid growth of system size and ease of doing so by upgrading the software.

Modeling with AIRS:

Once the detailed reliability models are complete, the reliability analyst must further decompose those system elements containing AIRS. For the purposes of reliability modeling, software includes AIRS, which is configurable or under configuration control.

A. Modeling series:

Failure rates for operating systems or executives, if available, can be obtained from the supplier of the operating system or executive. Failure rates obtained from the operating system supplier are usually quoted in the number of outages caused over some period of time (e.g., a year). Failure rates for operating systems are generally quoted with respect to system operating time because the operating system is active at all times when the computer is powered and ready for processing. The reliability analyst will need to convert the failure rate given to failures per hour for compatibility with hardware failure rates.

B. Flow Diagram:

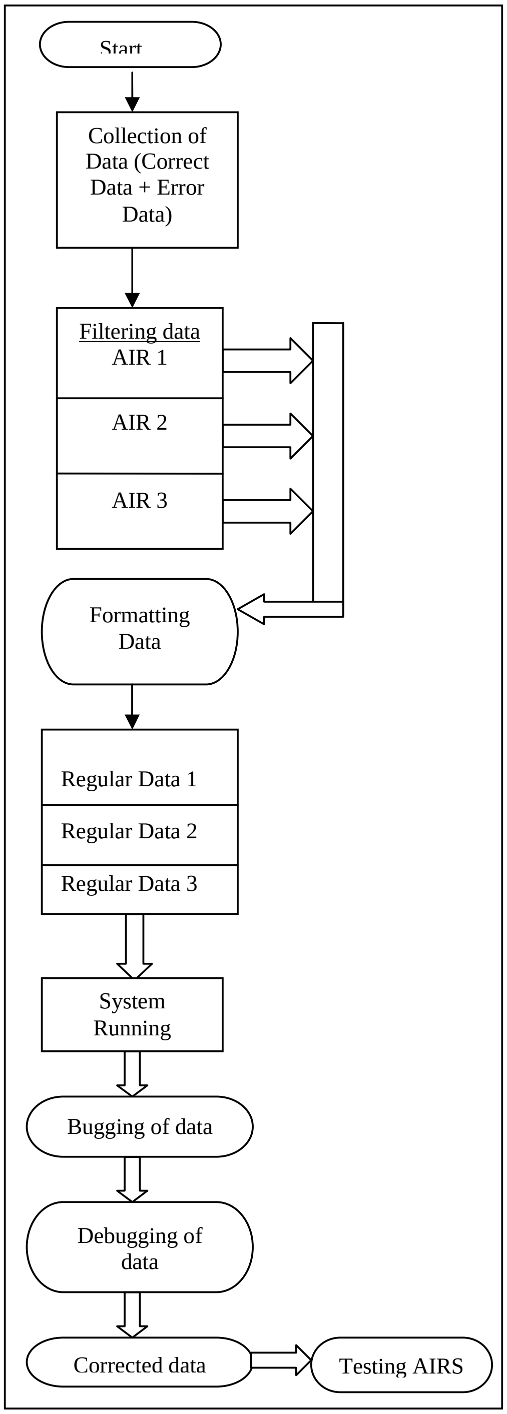

Fig.1

AIRS Algorithm:

I Initialization:

Collection of data to normalized with the required system.

II Filtering:

Filtering is nothing but the collected data wanted to be segregated by there respective cell allocation i.e. the set of processed data is called information. This information is reliable for our system.

III Definition of AIRS:

The Artificial Immune Recognition System (AIRS) is a new method for data mining [10,11].In this section we explore the application of the AIRS for software quality classification. We first prepare a pool or recognition or memory cells (data exemplars) which are the representatives of the training software modules the model is exposed to. The lifecycle of the AIRS is illustrated in Fig.2.

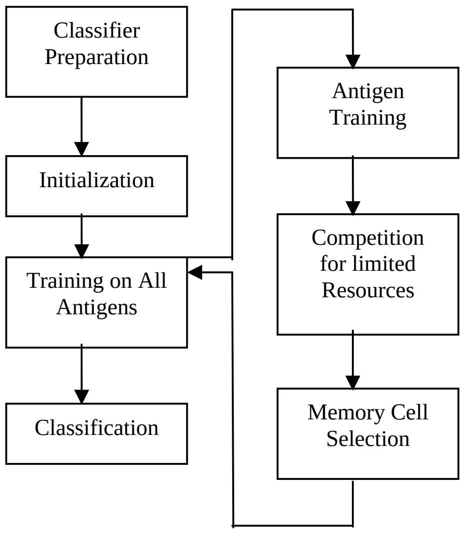

Fig.2

IV Classification:

The normalized regular formation of data is going to classify with the required AIRS with different groups of data.

V Software Failure Rate (SFR):

Determining software failure rates for use in combined hardware/software models requires that the software being analyzed be treated as a subsystem. A software subsystem, like hardware, can be viewed as a hierarchy. As far as reliability is concerned, however, the hierarchy consists of functions or operations rather than components. The software functions that comprise a system will be related to one another in two ways: a particular timing configuration and a particular reliability topology.

Timing configuration is a concern when the various functions are active and inactive during a period of interest. Topology concerns the number of functions in the system that can fail before the system fails.

SFR is measured by using following method based on mathematical technique.

Consider q level data, it contains the data with some failure (or) incorrect form

i.e. Dq = CDq-1+EDq-1

Where Dq = Data in q level

CDq-1 = Correct data in q-1 level

EDq-1 = Error data in q-1 level

Where D q+1=Dq-EDq-1

This will give corrected data in q level process. This is known as recurrence formula.

Conclusion:

In this paper we discussed only ground level net work of Software Failure Rate (SFR) for the Artificial Immune Recognition System (AIRS) with mathematical formula. In future definitely this method will help us to do further steps in different mathematical technique.

References:

1. S. Beattie, S. Arnold, C. Cowan, P.Wagle, and C. Wright, “Timing the application of security patches for optimal uptime,” in *Proc. LISA XVI*, November 2002, pp. 233–242.
2. B.Brykczynski and R. A. Small, “Reducing internet-based intrusions:Effective security patch management,” *IEEE Software*, vol. 20, no. ,pp. 50–57, Jan./Feb. 2003.
3. *“Handbook of Software Reliability Engineering,”* M. R. Lyu, Ed., Mc-Graw-Hill, 1995.
4. J. D. Musa, *Software Reliability Engineering*. : McGraw-Hill, 1999.
5. E. E. Schultz, Jr., D. S. Brown, and T. A. Longstaff, *Responding to Computer Security Incidents*. : Lawrence Livermore National Laboratory, July 23, 1990 .
6. J. Timmis, M. Neal, and J. Hunt, “An artificial immune system for data analysis,” Biosystems, vol. 55, no. 1, pp. 143–150, 2000.
7. M. Neal, “An artificial immune system for continuous analysis of time-varying data,” in 1st International Conference on Artificial Immune Systems, Canterbury, UK, 2002, pp. 76–85.
8. T Knight and J Timmis, “Aine: An immunological approach to data mining,” in IEEE International Conference on Data Mining, San Jose, CA, 2001, pp. 297–304.
9. S. Yacoub, B. Cukic, and H Ammar, “A Scenario-Based Reliability Analysis Approach for Component-Based Software,” *IEEE Transactions on Reliability*, vol. 53, no. 4, 2004, pp. 465-480.
10. .J.D. Musa, Software Reliability Engineering: More Reliable Software Faster and Cheaper (2nd Edition), AuthorHouse, 2004.X.

11. Teng, H. Pham, and D. Jeske, “Reliability Modeling of Hardware and Software Interactions, and Its Applications,” *IEEE Transactions on Reliability*, vol. 55, no. 4, Dec. 2006, pp. 571-577.
12. Farr, Dr. William, A Survey of Software Reliability Modeling and Estimation, NSWC TR 82-171, Naval Surface Weapons Center, Dahlgren, VA, Sept. 1983.
13. Friedman, M.A., Tran, P.Y., and Goddard, P.L., Reliability Techniques for Combined Hardware and Software Systems, Final Report, Contract F30602-89-C-0111, Rome Laboratory, Air Force Systems Command, Griffiss Air Force Base, New York. Sept. 1991.
14. Jones, Capers, Software Productivity Research, Inc., Applied Software Measurement, McGraw-Hill, NY, 1995.
15. Keene, Dr. Samuel, Cole, G.F., Reliability Growth of Fielded Software, Reliability Review, Vol 14, March 1994.
16. Lyu, Michael R., Handbook of Software Reliability Engineering, IEEE Computer Society Press, 1996.
17. a subsyst Musa, J.D., Iannino, A. and Okumoto, K., Software Reliability: Measurement, Prediction, Application, McGraw Hill Book Company, New York, NY. 1987.
18. H. Bersini and F. Varela, “Hints for Adaptive Problem Solving Gleaned from Immune Networks,” *in Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, Dortmund and Federal Republic of Germany, pp. 343-354 (1990).
19. D. Goodman, L. Boggess, and A. Watkins, “Artificial Immune System Classification of Multiple-Class Problems,” in C. H. Dagli, A. L. Buczak, J. Ghosh, M. J. Embrechts, O. Ersoy, and S. W. Kercel (eds.), *Intelligent Engineering Systems Through Artificial Neural Networks*, vol. 12, New York, NY, pp. 179-184 (2002).
20. S. Dick, A. Meeks, M. Last, H. Bunke, and A. Kandel, “Data Mining in Software Metrics Databases,” *Fuzzy Sets and Systems*, 145(1), pp. 81-110 (2004).
21. D. Garmus and D. Herron, *Measuring the Software Process*. Prentice Hall, Upper Saddle River, NJ (1996).
22. R. Subramanyan and M. S. Krishnan, “Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects,” *IEEE Trans. Software Eng.*, vol. 29, pp. 297-310, April (2003).
23. K. H. Muller and D. J. Paulish, *Software Metrics*, IEEE Press / Chapman & Hall, London, UK (1993).
24. NASA MDP, http://mdp.ivv.nasa.gov /index.html (2006).
25. A. Watkins and J. Timmis, “Artificial Immune Recognition System (AIRS): Revisions and Refinements,” in *Proceedings of the 1st International Conference on Artificial Immune Systems*, Canterbury, UK, pp. 173-181, September (2002).
26. A. Watkins, J. Timmis, and L. Boggess, “Artificial Immune Recognition System (AIRS): An Immune Inspired Supervised Machine Learning Algorithm,” *Genetic Programming and Evolvable Machines*, 5(1), March (2004).

[1] Prof .S.Chitra, is the Head of the Department of Computer Science and Engineering in M. Kumarasamy College of Engineering, Karur, India. She has 15 years experience in active teaching. She completed her BE and ME in Computer Science and Engineering and undergoing her research in the area Software Reliability . She presented more than 15 papers including national, international conferences and journals.

[2] Mr.K.Thiagarajan, Lecturer, Department of Mathematics, Rajalakshmi Engineering College, Thandalam, Chennai, India. He has attended and presented 23 papers in national and international conferences. He has published 13 international journals, and one National journal. He has 13 years of teaching experience.

[3] Dr.M.Rajaram, Professor of the department Electrical And Electronics Engineering in Thandhai perriyar government institute of technology, Vellore, India. He is guiding 12 research scholars and 4 have been awarded doctorate. He presented more than 100 papers including national, international journals and has 20 years of experience in teaching.