MEDWELL PUBLISHING
International Journal of Soft Computing 4 (2): 56-59, 2009
ISSN: 1816-9503

Problem Oriented Software Engineering: Estimating Software Package Reliability by Determining Error Rate

[1,2]O.O. Ekabua, [2]E.E. Williams and [2]O.A. Ofem
[1]Department of Computer Science, University of Zululand, South Africa
[2]Department of Maths, Statistics and Computer Science, University of Calabar, Nigeria

Abstract: Software packages play an important role in our lives. Products and services that affect people's lives must have quality attributes. Therefore, good quality software package is required and in order to determine the quality of software package we need methods to measure it reliability. An interesting point here is that the quality of software may change over time and software package is no exception. In the early days of computing, software costs represented a small percentage of the overall cost of a computer-based system. Hence, a sizable error in estimates of software cost had relatively little impact. Today, software is the most expensive element in many computer-based systems. Therefore, steps taken to reduce the cost of software can make the difference between the profit and loss of a company. So, by determining the quality attributes of software, more precise, predictable and repeatable control over the software development process and product will be achieved. We propose a formal approach from the problem oriented software engineering point of view, for estimating the reliability of a software package by determining the error rate.

Key words: Software estimate, software error, software reliability, software change, software fault, software package

INTRODUCTION

As an apparent statement, engineering is considered as a step-wise problem solving activity that yields a product (Krabbel *et al*., 1997). The use of problem-oriented method to requirements engineering is gradually becoming well established, mostly for software intensive systems. In recent years, the foundation of problem-oriented method is being laid (Rapanotti *et al*., 2004). Developing software has been viewed as a problem and the solution as a machine that is, a program running in a computer that will ensure satisfaction of requirement in the given problem world or environment. Because, the requirement typically concerns properties and behaviours that are located in the problem world at some distance from its interface with the machine, requirements are distinguished from specifications (Brier *et al*., 2004).

Amongst other concerns of software engineers is on how to handle change, which in most cases is inevitable. Changes occur as a result of fixing bugs, or adding and deleting requirement or functionality. Although, these are not the only sources of change, but the need for change is evolving as software packages evolves.

Very often, the terms errors, faults and failures are used interchangeable, but do have different meanings. In reference to software, an error is usually a programmer action or omission that results in a fault. While, a fault is a software defect that causes a failure and a failure refers to the unacceptable departure of a program operation from program requirements. When measuring reliability, we are usually measuring only defects found and defects fixed (Ekabua and Adigun, 2008).

Software failures may result from errors, ambiguities, oversights or misinterpretation of the specification that the software is supposed to satisfy, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems (Hall *et al*., 2002). While, it is tempting to draw an analogy between software reliability and hardware reliability, software and hardware have basic differences that make them different in failure mechanisms. Hardware faults are mostly physical faults, while software faults are design faults, which are harder to visualize, classify, detect and correct. Design faults are closely related to fuzzy human factors and the design process, which we don't have a solid understanding. In hardware, design faults may also exist, but physical faults usually dominate. In software, we can hardly find a strict corresponding counterpart for manufacturing as hardware manufacturing process, if the simple action of uploading

Corresponding Author: O.O. Ekabua, Department of Computer Science, University of Zululand, South Africa

software modules into place does not count. Therefore, the quality of software will not change once it is uploaded into the storage and start running. Trying to achieve higher reliability by simply duplicating the same software modules will not work, because design faults can not be masked off by voting (Michael, 1995).

Software packages are being rolled by the day into the market, some evolving from the modification of a previous package. Developing (Hall *et al*., 2005) a software package is engineering in the traditional sense, as it involves the creation of an artefact, which transforms the physical world by meeting recognised needs. This poses challenges that logic alone cannot address, but can be met only by sharply focused specialisms practicing highly developed normal design disciplines, in line with those that characterise the established engineering disciplines.

These being well known difficulties in established branches of engineering, have sometimes led to harmful dichotomy in approaches to software package development. Some approaches are address as formal concerns, while others as non-formal concerns-creating difficulties in reconciling the two.

A key challenge therefore, is for software engineers to develop methods on how to reconcile the formal world of the machine and its software with the non-formal real world. This is where Problem Oriented Software Engineering (POSE) becomes necessary. Unquestionably, engineering is a problem solving activity (Ekabua and Adigun, 2008). Our focus is not only on requirements engineering, where the problem-oriented software engineering has its origins, but more generally with software engineering leading to the development of a reliable software package. Along with other views (Hall *et al*., 2005), software engineering includes the identification and clarification of system requirements, the identification, structuring and analysis of the problem world, the structuring and specification of a hardware/software machine that can ensure satisfaction of the requirements in the problem world, the creation of the software product and the construction of formal arguments, convincing to developers, customers, users and other interested people, that the system will satisfy its real world requirements.

RELATED WORK

A formal conceptual framework for software development based on a problem-oriented perspective that stretches from requirements engineering through to program code was introduced. The research regarded development steps as transformations, by which problems are moved towards software solutions. The framework follows, the form of a sequent calculus, which can accommodate both formal and informal steps in development (Rapanotti *et al*., 2004).

Also, issues of producing and managing the safety reasoning involved in critical system development was addressed through POSE. In particular, the study provided some evidence on how POSE may contribute to those elements of a safety case arguing requirements validity and satisfaction, explicit context assumptions, design judgment and rationale and safety risk management and demonstrated the approach on a real world example (Hall *et al*., 2005).

OUR PROBLEM FRAME

As earlier mention, our problem domain is on estimating software package reliability. During the development of a new software package, a step-wise procedure is always put in place to eliminate earlier faults or bugs found in the package. An often implemented procedure is to allow the package to run on a set of well known problems to see if any errors result. This can go on for a fix time, while recording all resulting errors. After a while, the testing stops and the package carefully examine to determine the specific bugs responsible for the observed errors. Changes will now be effected in the package to remove these bugs.

Software reliability improvement is hard. The difficulty of the problem stems from insufficient understanding of software reliability and in general, the characteristics of software. Until, now there is no good way to conquer the complexity problem of software. Complete testing of a moderately complex software module is infeasible. Defect-free software product can not be assured. Realistic constraints of time and budget severely limits the effort put into software reliability improvement.

The POSE concept of problem requires a separation of context, requirement and clarification, with explicit descriptions of what is given, what is required and what is designed. This enhances the traceability of artifacts and their relation, as well as exposing all assumptions to scrutiny and validation. That all descriptions are generated through problem transformation forces the inclusion of an explicit justification that such assumptions are realistic and reasonable (Hall *et al*., 2005).

FORMAL PROBLEM FRAME TRANSFORMATION

A proliferation of software reliability models have emerged as people try to understand the characteristics of

how and why, software fails and try to quantify software reliability. Over 200 models have been developed since, the early 1970s, but how to quantify software reliability still remains largely unsolved. Interested readers may refer to Ekabua *et al*. (2007). As many models as there are and many more emerging, none of the models can capture a satisfying amount of the complexity of software; constraints and assumptions have to be made for the quantifying process. Therefore, there is no single model that can be used in all situations. No model is complete or even representative. One model may research well for a set of certain software, but may be completely off track for other kinds of problems.

Most software models contain the following parts: assumptions, factors and a mathematical function that relates the reliability with the factors. The mathematical function is usually higher order exponential or logarithmic. Software modeling techniques can be divided into 2 subcategories: prediction modeling and estimation modeling (Bleistein *et al*., 2004). Both kinds of modeling techniques are based on observing and accumulating failure data and analyzing with statistical inference.

Following our problem frame, it becomes possible to visualise that not all the bugs in the package have been eliminated and this introduces the great challenge of estimating the error rate of the revised software package.

To introduce a formal modelling method to the above scenario, we need to first suppose that the package initially contains an unknown number, M, of bugs, which herein is considered as bug 1, bug 2… bug M. Therefore, we make three assumptions as follows: First, we would like to assume that bug I will generate errors in accordance to a Poisson process with an unknown rate λ_i, i = 1. . . m. Then, for instance, the number of errors due to bug I that occur in any s units of operating time is Poisson distributed with mean λ_iS. Secondly, we will like to also, assume that these Poisson processes caused by bugs i, i = 1… M is independent. Thirdly, we assume the package is to be run for t time units with all resulting errors being noted. At the end of this period, debugging process takes place where a careful check of the package is made to determine the specific bugs that caused the errors. These bugs are removed and the problem is then to determine the error rate for the revised package.

If we let:

$$\psi_i(t) = \begin{cases} 0, \text{ if bug,i, has not caused an error by t} \\ 1, \text{ otherwise} \end{cases} \quad (1)$$

then

$$\Lambda(t) = \sum_i \lambda_i \psi_i(t) \quad (2)$$

is the error rate of the final package. The expectation of the above can then be stated as:

$$E\left[\wedge(t)\right] = \sum_i \lambda_i E\left[\varphi_i(t)\right] = \sum_i \lambda_i e^{-\lambda_i t} \quad (3)$$

We assume that each of the bugs discovered is responsible for a certain number of errors. Therefore, we denote M_i (t) as the number of bugs that were responsible for j errors, where, j≥1. That is M_i (t) caused exactly 1 error, M_i (t) caused exactly 2 errors and so on with Σ_j JM_j (t) equating the total number of errors that resulted. To compute E [M_1 (t)], we define I_i(t), where i≥1, by the relation:

$$\psi_i(t) = \begin{cases} 0, \text{bug i causes exactly 1error} \\ 1, \text{otherwise} \end{cases} \quad (4)$$

Then,

$$M_i(t) = \sum_i I_i(t) \quad (5)$$

and so

$$E\left[M_i(t)\right] = \sum_i E\left[I_i(t)\right] \quad (6)$$

$$= \sum_i \lambda_i t e^{-\lambda_i t} \quad (7)$$

Using Eq. 2 and 5, we obtain the intriguing result that:

$$E\left[\wedge(t) - M_1(t)/t\right] = 0 \quad (8)$$

From Eq. 6, we can possibly use M_1 (t)/t as an estimate of $\wedge$ (t) requires us to consider how far apart these two quantities tend to be. Therefore, we compute:

$$E\left[\left(\frac{\left(\wedge(t) - M_1(t)\right)}{t^2}\right)\right] = Var\left[\frac{\wedge(t) - M_1(t)}{t}\right] \quad (9)$$

$$= Var\left(\wedge(t)\right) - \frac{2}{t}Cov\left(\wedge(t), M_1(t)\right) + \frac{1}{t^2}Var\left(M_1(t)\right)$$

Now,

$$Var\left(\wedge(t)\right) = \sum_i \lambda_i^2 Var\left(\psi_i(t)\right) = \sum_i \lambda_i^2 e^{-\lambda_i t}\left(1 - e^{-\lambda_i t}\right) \quad (10)$$

$$Var\left(M_1(t)\right) = \sum_i Var\left(I_i(t)\right) = \sum_i \lambda_i t e^{-\lambda_i t}\left(1 - \lambda_i t e^{-\lambda_i t}\right) \quad (11)$$

$$\mathrm{Cov}\left(\wedge(t), M_1(t)\right) = \mathrm{Cov}\left(\sum_i \lambda_i \psi_i(t), \sum_j I_j(t)\right)$$

$$= \sum_i \sum_j \mathrm{Cov}\left(\lambda_i \psi_i(t), I_j(t)\right) \quad (12)$$

$$= \sum_i \lambda_i \mathrm{Cov}\left(\psi_i(t), I_i(t)\right) \quad (13)$$

$$= -\sum_i \lambda_i e^{-\lambda_i t} \lambda_i t e^{-\lambda_i t} \quad (14)$$

From Eq. 13 and 14, $\psi_i(t)$ and $I_j(t)$ are independent variables when $i \neq j$ because they refer to different Poisson processes and

$$\psi_i(t) I_j(t) = 0 \quad (15)$$

Therefore from Eq. 7, we obtain that

$$E\left[\left(\wedge(t) - \frac{M_1(t)^2}{t}\right)\right] = \sum_i \lambda_i^2 e^{-\lambda_i t} + \frac{1}{t}\sum_i \lambda_i e^{-\lambda_i t} \quad (16)$$

$$= \frac{E\left[M_1(t) + 2M_2(t)\right]}{t^2} \quad (17)$$

Following Eq. 17 and 5 and the identity, we have

$$E\left[M_2(t)\right] = \frac{1}{2}\sum_i (\lambda_i t)^2 e^{-\lambda_i t} \quad (18)$$

We can therefore, estimate the average square of the difference between $\wedge(t)$ and

$$\frac{M_i(t)}{t}$$

by the observed value of

$$\frac{\left(M_1(t) + 2M_2(t)\right)}{t^2}$$

CONCLUSION

The aim of the research reported here, is to bring both non-formal and formal aspects of software package development together in a step-wise formal mathematical models, so as to estimate software package reliability by determining error rate. The formal approach is intended to provide a structure within, which the results of different development activities can be combined and reconciled.

Essentially, the structure is the structure of the progressive solution of a system development problem; it is also, the structure of the argument that must eventually justify the adequacy of the developed software package. The model is itself formal, but it is designed to accommodate both formal and informal descriptions of problem domains and requirements and arguments justifying claimed relationships between development artifacts. Therefore, this study, has approached the problem of estimating software package reliability by determining error rate in software through formal mathematical methods.

REFERENCES

Brier, J., L. Rapanotti and J.G. Hall, 2004. Problem frames for socio-technical systems: Predictability and change. In: Proceedings of 1st Int. Workshop on Applic. Adv. Problem Frames, IEEE CS Press, pp: 21-25.

Bleistein, S., K. Cox and N. Verner, 2004. Problem Frames Approach for E-business Systems. In: Cox, K., J. Hall and L. Rapanotti (Eds.). 1st Int. Workshop Adv. Applic. Prob. Frames, Edinburgh, IEE., pp: 7-15.

Ekabua, O.O., O.O. Olugbara and M.O. Adigun, 2007. A generic change propagation framework to enhance service provisioning in a grid environment. Asian J. Inform. Technol., 6 (10): 1015-1019. www.medwell journals.com.

Ekabua, O.O. and M.O. Adigun, 2008. A framework and associated models for determining change impact analysis during utility service provisioning in a grid environment. Proceedings of the 2008 International Conference on Software Engineering Research and Practice (SERP). WORLDCOM, Las Vegas Nevada, USA. www.world-academy-of-science.org.

Hall, J.G., M. Jackson, R.C. Laney, B. Nuseibeh and L. Rapanotti, 2002. Relating Software Requirements and Architectures using Problem Frames. In 10th Anniversary IEEE Joint Int. Conf. Requirements Eng. (RE), Essen, Germany, IEEE. Comput. Soc., pp: 137-144.

Hall, J.G., L. Rapanotti and M. Jackson, 2005. Problem frame semantics for software development. J. Software and Syst. Modeling, 4 (2): 189-198.

Krabbel, A., I. Wetzel and H. Z"ullighoven. 1997. On the inevitable intertwining of analysis and design: developing systems for complex cooperations. In DIS: Proc. Conf. Designing Interactive Syst., New York, USA, 1997. ACM Press, pp: 205-213.

Michael, R.L., 1995. Handbook of Software Reliability Engineering. McGraw-Hill Publishing. ISBN: 0-07-039400-8.

Rapanotti, L., J.G. Hall, M. Jackson and B. Nuseibeh, 2004. Architecture-driven problem decomposition. In 12th IEEE Int. Conf. Requirements Eng. (RE 2004), IEEE Comput. Soc., pp: 80-89.