System Software Reliability

van Driel, Willem D.; Bikker, J.W.; Tijink, M.

DOI
10.1109/EuroSimE48426.2020.9152686

Publication date
2020

Document Version
Accepted author manuscript

Published in
2020 21st International Conference on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems (EuroSimE)

System Software Reliability

Willem D. van Driel[1], J.W. Bikker, M. Tijink
[1]Signify Eindhoven / Delft University of Technology, The Netherlands
[2]CQM, Eindhoven, The Netherlands
willem.van.driel@signify.com

Abstract

It is known that quantitative measures for the reliability of software systems can be derived from software reliability models. And, as such, support the product development process. Over the past four decades, research activities in this area have been performed. As a result, many software reliability models have been proposed. It was shown that, once these models reach a certain level of convergence, it can enable the developer to release the software. And stop software testing accordingly. Criteria to determine the optimal testing time include the number of remaining errors, failure rate, reliability requirements, or total system cost. In this paper we will present our results in predicting the reliability of software for agile testing environments. We seek to model this way of working by extending the Jelinski-Moranda model to a 'stack' of feature-specific models, assuming that the bugs are labelled with the feature they belong to. In order to demonstrate the extended model, several prediction results of actual cases will be presented. The questions to be answered in these cases are: how many software bugs remain in the software and should one decide to stop testing the software?

1. Introduction

Nowadays the lighting industry experiences an exponential increasing impact of digitization and connectivity of its lighting systems [1, 2]. The impact is far beyond the impact on single products, but extends to an ever larger amount of connected systems. Continuously, more intelligent interfacing with the technical environment and with different kind of users is being built-in by using more and different kind of sensors, (wireless) communication, and different kind of interacting or interfacing devices, see Figure 1.

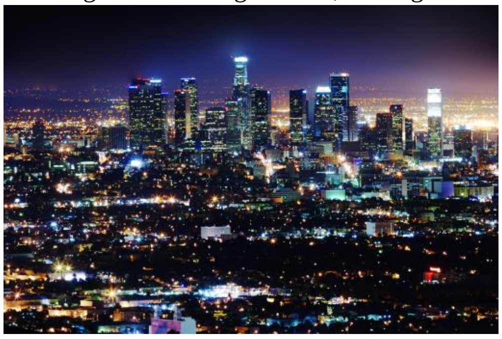

Figure 1: The growing population with increased urbanization results in the need to focus on energy efficiency and sustainability thereby increasing digitalization and rapidly evolving technologies containing software.

The trend towards controlled and connected systems also implies that other components will start playing an equal role in the reliability of such systems. Here, reliability needs to be complimented with availability and other modelling approaches are to be considered [3]. In the lighting industry, there is a strong focus on hardware reliability, including going from component reliability to system reliability. However, in the controlled and connected systems, software plays a much more prominent role than in even sophisticated "single" products such as color-adjustable lamps at home, streetlights, UV sterilization lights and alike. In these systems, availability is more strongly determined by software reliability than by hardware reliability [3]. In a previous study, the reliability of software was evaluated using the Goel-Okumoto reliability growth model [4]. It is known that different models can produce very different answers when assessing software reliability in the future [5]. A significant amount of research has been performed in the area of reliability growth and software reliability, that considers the process of finding (and repairing) bugs in existing software, essentially during a test phase [6 - 11]. A typical assumption is that the development of the software has finished, except for the bugs that have to be detected and repaired [5, 8, 12]. The software reliability models then answer questions such as: what is the number of remaining bugs?, how many would we find if we spend a specified number of additional weeks of testing, etc. [13, 14]. In a more recent study Rana et al. [15] demonstrated the use of eight different software reliability growth models that were evaluated on eleven large projects. Prior classification of the expected shape was proven to improve the software reliability prediction.

In many software developments companies, software is developed in a cadence of sprints resulting in biweekly releases in the so-called Scaled Agile Framework (SAFe) [16]. This means there is a second reason why bugs are found, apart from finding them by doing tests, namely, new bugs are introduced because new features are added to the software continuously. An important class of software reliability growth models is known as General Order Statistics (GOS) models [17, 18]. The special case in which the order statistics come from an exponential distribution is known as the Jelinski-Moranda model [19]. The main assumption for this class of models is that the times between failures of a software system can be defined as the differences between two consecutive order statistics. It is assumed that the initial number of failures, denoted by a, is unknown but fixed and finite. In this paper, we seek to model this way of working by extending the Jelinski-Moranda model to a ‘stack’ of feature-specific models, assuming that the bugs are labelled with the feature they belong to. The feature-specific model parameters can be considered as random effects, so that differences between features are modelled as well. In order to demonstrate the extended model, two use cases will be presented. Here, we model the software testing phase to get a detailed sense of the software maturity. Once software is deemed mature enough by the organization, it is released to the end-users. The new, operational use of the software is different from testing phase, and this phase is not being modelled. The questions to be answered in the two cases are: how many software bugs remain in the software and should one decide to stop testing the software [20, 21]? This paper builds up the mathematical model that describes the number of bugs detected in every time interval (sprint), specified per software feature. We derive a way to evaluate the likelihood function, which is used in the next section on estimation. We set out with the model with only one feature, which is a variant of the Jelinski-Moranda model but adapted for the counts per sprint. We need expressions for conditional probabilities based on recent history, where only the cumulative counts turn out to be important. We extend the results to multiple features, where we shift the time axis as different software features are completed at different times. We conclude by describing how all ingredients are combined to the likelihood function.

2. Mathematical derivations and approach

Full details for the mathematical derivations can be found in [22]. The basic concept includes that a software tool has bugs, which are detected at time Ti after testing starts at time 0. T_i is independent and exponentially distributed, i.e., Individual bugs are found independently following an exponential distribution. To model agile software development, where new functionality is added after each sprint (taking say two working weeks), we consider software as a set of features: one feature can be considered a single part of the software, or the result of a single “sprint” of development. Bugs are found and fixed for the existing features (the latest and earlier features), and new features can be added at later points in time. This way, you can track and predict the remaining number of bugs for the current set of features (or any other interesting set of features).

The bug reports may come from different sources (implemented regression tests and tests by the team). Only bugs of sufficient severity are considered in the predictions. To handle the various sources we simply took the aggregate counts per sprint as input, assuming that the total number of tests in a sprint was comparable, we get a discrete time axis that was reasonably close to both test effort and calendar time. Ticket data were fed into the code, where we distinguished tickets with severity levels S (high) and A (low). We used JIRA [23] output of bug data, a typical out is shown in Figure 2. Pick-and-mix was used for ticket severity allocation. These tickets either had the allocation open or closed. Open means the issues were being solved, closed means it was solved. Recurring tickets were treated as a new open ticket which can be closed as soon as it is known to be recurrent. Ticket severity is denoted as S, A, B, C, or D. S are issues seen as a blocker that need immediate attention. A is seen as critical, B as major C and D as minor severity levels. We have only analyzed the closed tickets. Figure 2 depicts the full flowchart of the process: from tickets to dashboard values. Actual sprint dates have an equal length for each sprint of two weeks. The outcome is produced automatically.

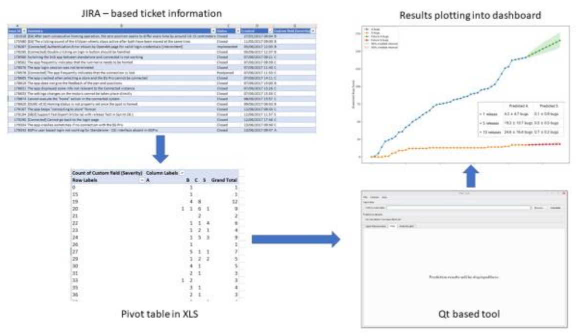
JIRA – based ticket information
Results plotting into dashboard
Pivot table in XLS
Qt based tool

Figure 2: Flowchart for automatic generation of software reliability predictions.

3. Results

We analyzed 8 system projects with the developed tool. All these projects are still in the development phase and follow clear software quality principles. In total, it concerns approximately 10.000 software tickets or bugs. Figure 3 depicts the ticket distributions when classified as high (A + S tickets) and low (B + C + C) tickets. The variation per project is clear, tickets classified as high cover approximately 12% of all, and low about 88%. This was to be expected as severe tickets should appear less then less severe ones.

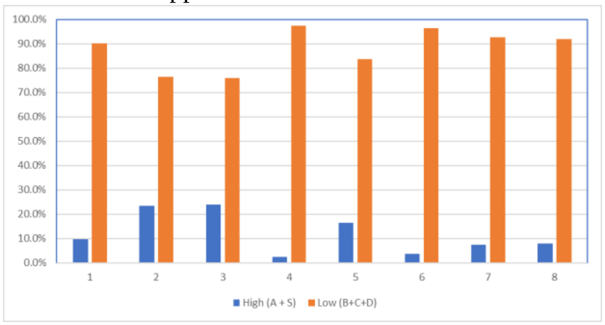

Figure 3: Ticket distributions for the 8 analyzed projects.

Predicted results of 4 projects, 1, 4, 6 and 7, are depicted in Figure 4. It shows the cumulative growth of severe (orange – red) and less severe (blue – green) tickets as function of sprints (in this case weeks). For projects 4 and 6 no signs of maturity is near, for projects 1 and 7, maturity is in sight. The predicted data is shown in Table 1. This table depicts the average values of predicted nr of tickets in coming sprints. Some projects are seeing maturity that are those with a low nr of remaining bugs after 10 sprints such as project 1. Most projects are seeing good levels of maturity for high severity tickets. Project 5 is the exemption, with still a large amount of severe tickets remaining in the code. Again, all projects are still in the development stage. The predicted values presented in Table 1 can serve for decisions to be taken if the software can be launched into the market. Also, this data can be used to allocate manpower for further code development and/or enhancement. Question remains for all these projects: can we take that decision?

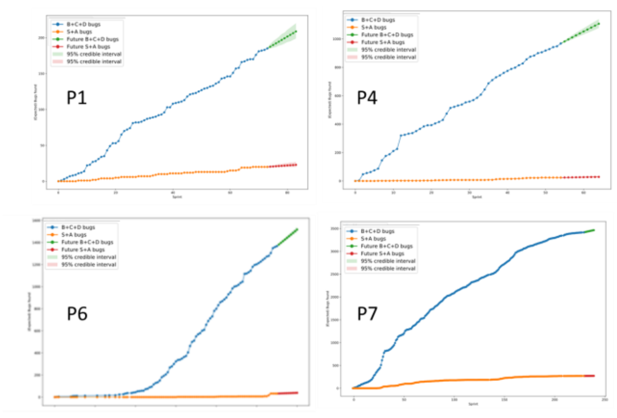

Figure 4: Predicted tickets as function of sprints (weeks) for projects 1, 4, 6 and 7. Blue lines concerns low severity tickets orange lines the high ones. Future tickets are given in green and red.

Table 1: Predicted nr of tickets for coming sprints. Average values +/- standard deviation.

Project	Predicted nr of tickets			
	High (A + S)		**Low (B + C + D)**	
	+1 sprint	+10 sprints	+1 sprint	+10 sprints
1	0.3 +/- 1.7	1.4 +/- 2.6	2.4 +/- 3.6	11.9 +/- 8.1
2	0.7 +/- 2.3	3.2 +/- 4.8	2.6 +/- 3.4	12.7 +/- 8.3
3	1.0 +/- 3.0	5.1 +/- 5.9	5.2 +/- 4.8	25.8 +/- 11.2
4	0.5 +/- 1.5	2.4 +/- 3.6	14.7 +/- 7.8	71.0 +/- 20.0
5	3.9 +/- 4.1	19.3 +/- 9.7	8.1 +/- 5.9	39.9 +/- 14.1
6	0.6 +/- 1.4	2.9 +/- 4.1	15.0 +/- 8.0	75.2 +/- 18.8
7	0.2 +/- 0.8	0.8 +/- 2.2	5.2 +/- 4.8	25.1 +/- 10.9
8	0.2 +/- 0.8	1.0 +/- 2.0	2.2 +/- 3.8	10.5 +/- 7.5

4. Discussion & conclusions

Software failures differ significantly from hardware failures. They are not caused by faulty components or wear-out due to e.g. physical environment stresses such as temperature, moisture and vibration. Software failures are caused by latent software defects. These defects were introduced in the software while it was created. However, these defects were not detected and/or removed prior of being released to the customer. In order to prevent that these defects are noticed by the customer, a higher level of software reliability has to be achieved. This means to reduce the likelihood that latent defects are present in released software. Unfortunately, even with the most highly skilled software engineers following industry best practices, the introduction of software defects is inevitable. This is due to the ever-increasing inherent complexities of the software functionality and its execution environment. Here, software reliability engineering may be helpful, a field that relates to testing and modelling of software functionality in a given environment of a particular amount of time. But certainly, there is currently no method available that can guarantee a totally reliable software. In order to achieve the best possible software, a set of statistical modelling techniques are required that:

- Can assess or predict the to-be-achieved reliability;
- Based on the observations of software failures during testing and/or operational use.

In order to achieve these two requirements, many software reliability models have been proposed. It was shown that, once these models reach a certain level of convergence, it can enable the developer to release the software. And stop software testing accordingly. Criteria to determine the optimal testing time include the number of remaining errors, failure rate, reliability requirements, or total system cost. Typical questions that need to be addressed are:

- How many errors are still left in the software?
- What is the probability of having no failures in a given time period?
- What is the expected time until the next software failure will occur?
- What is the expected number of total software failures in a given time period?

Certainly, the question on “How many errors are left” is something completely different from “What is the expected number of errors in a given time period”. One cannot estimate the first directly, but you can estimate the second. In our approach, we are content with “expected number of errors that a long testing period would yield”.

In this paper we presented an approach to predict software reliability for agile testing environments. The new approach divers from the many others in the sense that it combines features with tickets using Bayesian statistics. By doing that, a more reliable number of predicted tickets (read: software bugs) can be obtained. The developed system software reliability approach is applied to 8 software development projects, to demonstrate how software reliability models can be used to improve the quality metrics. The new approach is carved down in a tool, programmed in Python. The outcome of the predictions can be used in the Quality dashboard maturity grid to enable a better judgement of releasing the software or not. The strength of the software reliability approach is to be proven by more data and comparison with field return data. The outcome is satisfactory as a more reliable number of remaining tickets was calculated. As prominent advantage we note that divergence of the proposed fitting procedure is not an issue anymore in the new approach.

Following is recommended for the future developments of the presented approach:

- Gather more data from the software development teams.
- Connect to the field quality community to gather field data of software tickets.
- Make software reliability calculation part of the development process
- Automate the Python code such that ticket-feature data can be imported on-the-fly.
- Include machine learning techniques and online failure prediction methods, which can be used to predict if a failure will happen 5 minutes from now [24].
- Investigate the used of other SRGM models, including multistage ones, or those that can distinguish development and maintenance software defects [14, 15].
- Not focus on a specific software reliability model but rather assess forecast accuracy and then improve forecasts as was demonstrated by Zhao et al [25].
- Classify the expected shape of defect inflow prior to the prediction [15].

Acknowledgments

The European project SCOTT is acknowledged for funding the presented work in this paper.

References

1. Van Driel, W.; Fan, X. Solid State Lighting Reliability: Components to Systems; Springer: New York, 2012. 359 doi:10.1007/978-1-4614-3067-4.
2. Van Driel, W.; Fan, X.; Zhang, G. Solid State Lighting Reliability: Components to Systems Part II; Springer: New York, 2016. doi:10.1007/978-3-319-58175-0.
3. Papp, Z.; Exarchakos, G., Eds. Runtime Reconfiguration in Networked Embedded Systems - Design and Testing Practice; Springer: Singapore, 2016. doi:doi: 10.1007/978-981-10-0715-6.
4. Van Driel,W.; Schuld, M.;Wijgers, R.; Kooten,W. Software reliability and its interaction with hardware reliability. 15th International Conference on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems (EuroSimE), 2014.
5. Abdel-Ghaly, A.A.; Chan, P.Y.; Littlewood, B. Evaluation of competing software reliability predictions. IEEE Trans. Softw. Eng. 1986, SE-12, 950–967.
6. Bendell, A.; Mellor, P., Eds. Software Reliability: State of the Art Report; Pergamon Infotech Limited: Maidenhead, 1986.
7. Lyu, M., Ed. Handbook of Software Reliability Engineering; McGraw-Hill and IEEE Computer Society: New York, 1996.
8. Pham, H., Ed. Software Reliability and Testing, Los Alamitos, California, 1995. IEEE Computer Society Press.
9. Xie, M. Software reliability models—past, present and future. In Recent advances in reliability theory, Bordeaux, 2000; Stat. Ind. Technol., Birkhäuser Boston: Boston, MA, 2000; pp. 325–340.
10. Bishop, P.; Povyakalo, A. Deriving a frequentist conservative confidence bound for probability of failure per demand for systems with different operational and test profiles. Reliability Engineering & System Safety 378 2017, 158, 246–253.
11. Adams, E. Optimizing preventive service of software products. IBM Journal of Research and Development 380 1984, 28, 2–14.
12. Xie, M.; Hong, G. Software reliability modeling, estimation and analysis. In Advances in Reliability; North-Holland: Amsterdam, 2001; Vol. 20, Handbook of Statist., pp. 707–731.
13. Almering, V.; Van Genuchten, M.; Cloudt, G.; Sonnemans, P. Using Software Reliability Growth Models in Practice. Software, IEEE 2007, 24, 82–88.
14. Pham, H., Ed. System Software Reliability; Springer-Verlag: London, 2000. doi:10.1007/1-84628-295-0.
15. Rana, R.; Staron, M.; Berger, C.; Hansson, J.; Nilsson, M.; Törner, F.; Meding, W.; Höglund, C. Selecting pm nhu8io0software reliability growth models and improving their predictive accuracy using historical projects data, Journal of Systems and Software 2014, 98, 59–78.
16. Xie, M.; Hong, G.; Wohlin, C. Modeling and analysis of software system reliability. In Case Studies in Reliability and Maintenance; Blischke, W.; Murthy, D., Eds.; Wiley: New York, 2003; chapter 10, pp. 233–249.
17. Miller, D. Exponential order statistic models of software reliability growth. IEEE Transactions on Software, Engineering 1986, SE-12, 12–24.
18. Joe, H. Statistical Inference for General-Order-Statistics and Nonhomogeneous-Poisson-Process

Software Reliability Models. IEEE Trans. Software Eng. 1989, 15, 1485–1490.

19. Jelinski, Z.; Moranda, P. Software Reliability Research. In Statistical Computer Performance Evaluation; Freiberger, W., Ed.; Academic Press, 1972; pp. 465–497.
20. Dalal, S.R.; Mallows, C.L. When should one stop testing software? J. Amer. Statist. Assoc. 1988, 83, 872–879.
21. Zacks, S. Sequential procedures in software reliability testing. In Recent advances in life-testing and reliability; CRC: Boca Raton, FL, 1995; pp. 107–126. Version April 21, 2020 submitted to Mathematics.
22. W.D. van Driel, J.W. Bikker, M Tijink, A. Di Bucchianico, Software Reliability for Agile Testing, Accepted for publication in Mathematics, 2020.
23. Atlassian. JIRA Software Description, 2020.
24. Salfner, F., L.M..M.M. A survey of online failure prediction methods. ACM Computing Surveys 2010, 433 42, 12–24.
25. Zhao, X.; Robu, V.; Flynn, D.; Salako, K.; Strigini, L. Assessing the Safety and Reliability of Autonomous Vehicles from Road Testing. 30th International Symposium on Software Reliability Engineering (ISSRE) 436 2019, 2019.