Improving the Accuracy of Software Reliability Modeling by Predicting the Number of Secondary Software Defects

Oleg Odarushchenko[a], Elena Odarushchenko[b], Olena Kopishynska[c], Oleksandr Rudenko[d] and Anatoliy Gorbenko[e]

[a] *Poltava State Agrarian University, odarushchenko@gmail, Poltava, Ukraine*
[b] *Poltava State Agrarian University, elena.odarushchenko@gmail, Poltava, Ukraine*
[c] *Poltava State Agrarian University, olena.kopishynska@pdaa.edu.ua, Poltava, Ukraine*
[d] *National University «Yuri Kondratyuk Poltava Polytechnic», olexantr@gmail.com, Poltava, Ukraine*
[e] *Leeds Beckett University, a.gorbenko@leedsbeckett.ac.uk, Leeds, United Kingdom*

Abstract

Reliability assessment and prediction of the number of faults/defects is an important part of the software engineering process. Many software reliability models assume that all detected are removed with certainty and no new faults are introduced. However, the introduction of secondary faults during software updates has become quite common in software development practice, which can be explained by the enormous complexity of modern computer applications. In the paper we consider different scenarios of introducing secondary faults and how to predict number of such faults. Finally, we discuss how different SRGMs like Jelinski-Moranda, Exponential, Schick-Wolverton, Musa and Lipov models can be modified to account secondary faults in order to improve accuracy of software reliability prediction. We use an industrial case study to demonstrate applicability of the proposed approach. Our results show that considering secondary faults helped to considerably improve accuracy of software failure rate prediction.

Keywords

Software reliability, software reliability growth models, secondary faults, modified Jelinski-Moranda model

1. Introduction

Software (SW) has become a ubiquitous component and an important part of modern information and communication systems. As a result, software failures caused by faults made during software development and overlooked during the testing process can have severe consequences and lead to large-scale outages of computer systems, significant financial losses and even human casualties.

Thus, assessment of software reliability is one of the key issues that arise during SW development, verification, and validation. An importance of software reliability assessment is emphasized by the need to comprehensively account its' impact on reliability of computer systems for critical and busines-critical applications [1, 2]. Quantitative approach to evaluate software reliability is based on application of various mathematical models assessing and predicting the number of residual faults and probability of failure occurrence. These models are called software reliability growth models (SRGMs) [3-5]. SRGMs reflect the critical difference in software and hardware failure mechanisms (hardware fails mostly due to physical faults, while software faults are design faults; reliability of hardware can degrade

IntelITSIS'2022: 3rd International Workshop on Intelligent Information Technologies and Systems of Information Security, March 23–25, 2022, Khmelnytskyi, Ukraine
EMAIL: odarushchenko@gmail (A. 1); elena.odarushchenko@gmail (A. 2); olena.kopishynska@pdaa.edu.ua (A. 3); olexantr@gmail.com (A. 4); a.gorbenko@leedsbeckett.ac.uk (A. 5)
ORCID: 0000-0003-3933-9637 (A. 1); 0000-0002-2293-2576 (A. 2); 0000-0002-3138-7215 (A. 3); 0000-0002-1847-6635 (A. 4); 0000-0001-6757-1797 (A. 5)

over time due to ageing, while software cannot age physically) and can predict the mean time to failure (MTTF) based on failure statistics collected during software testing and operation.

A large number of software reliability models have been proposed over the past decades. A model is a description of the relationship between two or more variables (e.g. the number of already detected faults/defects and the time to the next failure).

Model assumptions are used to simplify model development. They denote a set of used conventions, choices and other specifications on which the model is based. Many SRGMs contain an assumption that in the process of eliminating already detected (primary) faults, new (secondary) faults cannot be introduced [8]. However, the practice of software engineering shows that secondary faults/defects are often introduced during software update which needs to be accounted by SRGMs [2]. This can be done by developing a new reliability model from the scratch or via modification/upgrading of the existing SRGMs by introducing new thoughtful assumption [6].

The paper proposes a new technique for prediction the number of secondary faults. This technique has been incorporated into Jelinski-Moranda SRGM to improve accuracy of software reliability assessment. An industrial case study was used to justify applicability of the proposed approach and to demonstrate its efficiency.

2. Related Works

Over two hundred different software reliability models have been developed over the past decades. These models define the set of assumptions about the number of faults (finite or infinite), fault rate, time between failures, etc. and offer analytical equations forecasting the number of residual failures, the failure rate and mean time to failure. A large set of different assumptions is caused by the wide range of different factors in the process of software development, testing and operation affecting conditions of faults creation, detection and manifestation [14-23].

There have been quite a few publications attempting to systematize and classify existing SRGMs based on the assumptions they use:

- *Hecht's classification* [7] divided models into *prediction*, *estimation* and *measurement* models; *prediction* models such as Holsted and Motley-Brook predict number of software defects based on physical program characteristics, i.e. code metrics, such as number of the source lines of code, number of cycles and cyclomatic complexity, number of errors per page of the program code, etc.; *measurement* models (e.g. Nelson-Bastani) evaluate reliability of a program working in an operational environment for a sample of inputs which are randomly selected from the whole input domain set and counting the number of inputs that results in execution failures; *estimation* models like Musa model predict mean time to failure based on failure statistics collected during software testing.
- *Goel's classification* [8] considered the following model domains: *Times Between Failures Models* (e.g. Jelinski-Moranda and Schick-Wolverton models); *Failure Count Models*: (e.g. Schumann model); *Fault Seeding Models*: (e.g. Mills and Beisin models); *Input Domain Based Models* (e.g. Nelson model).
- *Fatuev's classification* [8]. According to Fatuev's classification SRGMs can be divided into two major classes: *static* and *dynamic*. In turn, each class can be split into *continuous* and *discrete* subclasses.
- *Blagodatsky's classification* [9] extended Fatuev's classification by further dividing the static models by *defects* and *input data domains* and introducing a new class of *empirical* models which encompasses the complexity model and the model that determines the required software debugging time. Continuous models include: Jelinski-Moranda, Mousses, and Transition probabilities models. The class of discrete models consists of: Shumkan, La Padula, and Schick-Wolverton SRGMs.
- *Polonnikov-Nikandrov* [10] divided models by time structure (Jelinski-Moranda, simple exponential, Schick-Wolverton, Lipov, geometric, Schneidewind, Weibul, and Duane models), software complexity (Halsted model), error markup (Mills, Beisin, and simple heuristic models), program text structure (Nelson, La Padula, and IBM regression models), input data space structure (text and entropic models).

- *Kharchenko* et al in [24] put forward a facet-hierarchical classification of the most popular SRGMs (Jelinski-Moranda, Shooman, Goel-Okumoto, Schneidewind, Musa, Lapri, Lipow, Musa-Okumoto etc.) which is based on their assumptions systematisation and offered a method of software reliability growth models choice using the proposed assumptions matrix.

Ultimately, we can conclude that software reliability models can be divided into three main classes: (i) empirical models, (ii) statistical models and (iii) probabilistic models. The class of probabilistic models seems to be the most suitable to consider and take into account probability of inserting the secondary faults during the process of removing the primary faults. We selected a subset of probabilistic models which risk functions can be modified to account probability of inserting the secondary faults: Jelinski-Moranda, Simple exponential, Schick-Wolverton, Lipov and Musa models. In the next section we will consider scenarios of introducing secondary defects during the software life cycle and how the risk functions of the software reliability models mentioned above can be updated to account for secondary faults in order to improve the accuracy of software reliability predictions.

3. Scenarios of introducing SW secondary faults

It is assumed that software contains N_d initial faults and undergoes a series of updates/upgrades during its lifecycle. The number of faults M_i left in the software after the *i*-th update/upgrade can be estimated using (1) depending on one of the following assumptions:

1. All faults N_i are removed with certainty and no new faults are introduced;
2. All faults N_i detected by the *i*-th update step are removed with certainty; however, K_i new (secondary) faults are introduced after software is updated;
3. Not all faults out of N_i detected by the *i*-th update step are removed after updating the software; ΔN_i faults are left unfixed, but no new faults are introduced as a result of software update;
4. Not all faults out of N_i detected by the *i*-th update step are removed after updating the software; ΔN_i faults are left unfixed; in addition, K_i new (secondary) faults are introduced as a result of software update;
5. All faults N_i detected by the *i*-th update step are removed with certainty; however, K^*_i new faults are introduced after upgrading software functionality (i.e. adding new functions);
6. Not all faults out of N_i detected by the *i*-th update step are removed after updating the software; ΔN_i faults are left unfixed; in addition, K^*_i new faults are introduced after upgrading software functionality (i.e. adding new functions);
7. All faults N_i detected by the *i*-th update step are removed with certainty; however, K_i* new faults are introduced after upgrading software functionality (i.e. adding new functions); in addition, K_i^B interaction faults (i.e. faults occurred during interaction of upgraded software modules with non-upgraded ones) are introduced into the program.
8. Not all faults out of N_i detected by the *i*-th update step p are removed with certainty; however, K_i* new faults are introduced after upgrading software functionality (i.e. adding new functions); in addition, K_i^B interaction faults (i.e. faults occurred during interaction of upgraded software modules with non-upgraded ones) are introduced into the program.

$$M_i = \begin{cases} N_d - N_i, \\ N_d - N_i + K_i, \\ N_d - N_i + \Delta N_i, \\ N_d - N_i + K_i + \Delta N_i, \\ N_d - N_i + K_i^*, \\ N_d - N_i + \Delta N_i + K_i^*, \\ N_d - N_i + K_i^* + K_i^B, \\ N_d - N_i + K_i^* + K_i^B + \Delta N_i, \end{cases} \tag{1}$$

Table 1 summarizes updates which need to be made in the risk functions of different software reliability models to incorporate new assumptions.

Table 1
Modified risk functions

SRGM	Risk Functions	Modified risk functions
Jelinski-Moranda	$\lambda(t)=K(N_d - (i-1))$	$\lambda(t)=K(N_d-i+1+n^{in})$
Simple exponential	$\lambda(t)=K(N_d -N(t))$	$\lambda(t)=K(N_d - N(t)+n^{in})$
Schick-Wolverton	$\lambda(t)=K(N_d - i+1)X_i$	$\lambda(t)=K(N_d - i+1+n^{in})X_i$
Lipov	$\lambda(t)=K(N_d - F_{i-1})$	$\lambda(t_i)=K(N_d - F_{i-1}+n^{in})$
Musa	$\lambda(t)=K(N_d - F_{i-1})$	$\lambda(t_i)=K(N_d - F_{i-1}+n^{in})$

where $\lambda(t)$ – risk functions, F_{i-1} – the total number of corrected defects at the time, X_i – test time from t_i-1 (time of detection of i-1 software defect) to t_i.

4. Software reliability assessment workflow

The process of predicting the number of secondary SW faults can be split into the following steps.

Step 1. Collecting the SW defects statistics.

As an input this step uses the following information [11-13]:

- system requirements specification (SRS): description of system functions, operating scenarios, interfaces description, functional and non-functional (i.e. performance, environment, information security, reliability) requirements;
- software requirements specification (SWRS) and software detailed design (SWDD): description of operating SW scenarios, SW functional requirements and requirements for SW interfaces, and non-functional requirements;

The output of this stage is statistics about failure occurrence and fault detection. It is obtained based on the analysis of testing and validation reports and code reviews.

Step 2. Analyzing failure occurrence and fault detection statistics as a function of time.

Failure occurrence and fault detection statistics in each time interval collected at the previous stage is used to analyze its time correlation.

Step 3. Selecting the regression function.

Step 4. Evaluating regression coefficients.

Step 5. Predicting the number of secondary SW faults.

4.1. Prediction of the number of secondary software faults

Below we define steps for predicting the number of secondary SW defects.

1. Calculating the module of the difference between the actual number of software defects detected in the specified time interval and the value predicted by the regression function for the same period of time.

2. The number of secondary SW faults in the i-th testing interval can be estimated as the difference between the results obtained in the previous step and the standard deviation of the fault statistic in the specified time interval multiplied by the coefficient (2) and rounded to the nearest whole number.

$$\frac{1}{n+1-i}, \tag{2}$$

where n – is the total number of testing intervals.

As a result, we can derive the following equation:

$$\delta = \left| y - \frac{a}{x} - b \right| - \frac{1}{n+1-x}\sigma_y, \tag{3}$$

where y – is statistics of software faults detected in each testing interval; x – is the sequence number of the testing interval; a, b – are coefficients of the regression function; n – is the total number of testing intervals; σ_y – is the standard deviation of y; δ – is the estimated deviation value.

The number of secondary faults then can be estimated by rounding δ to the nearest whole number.

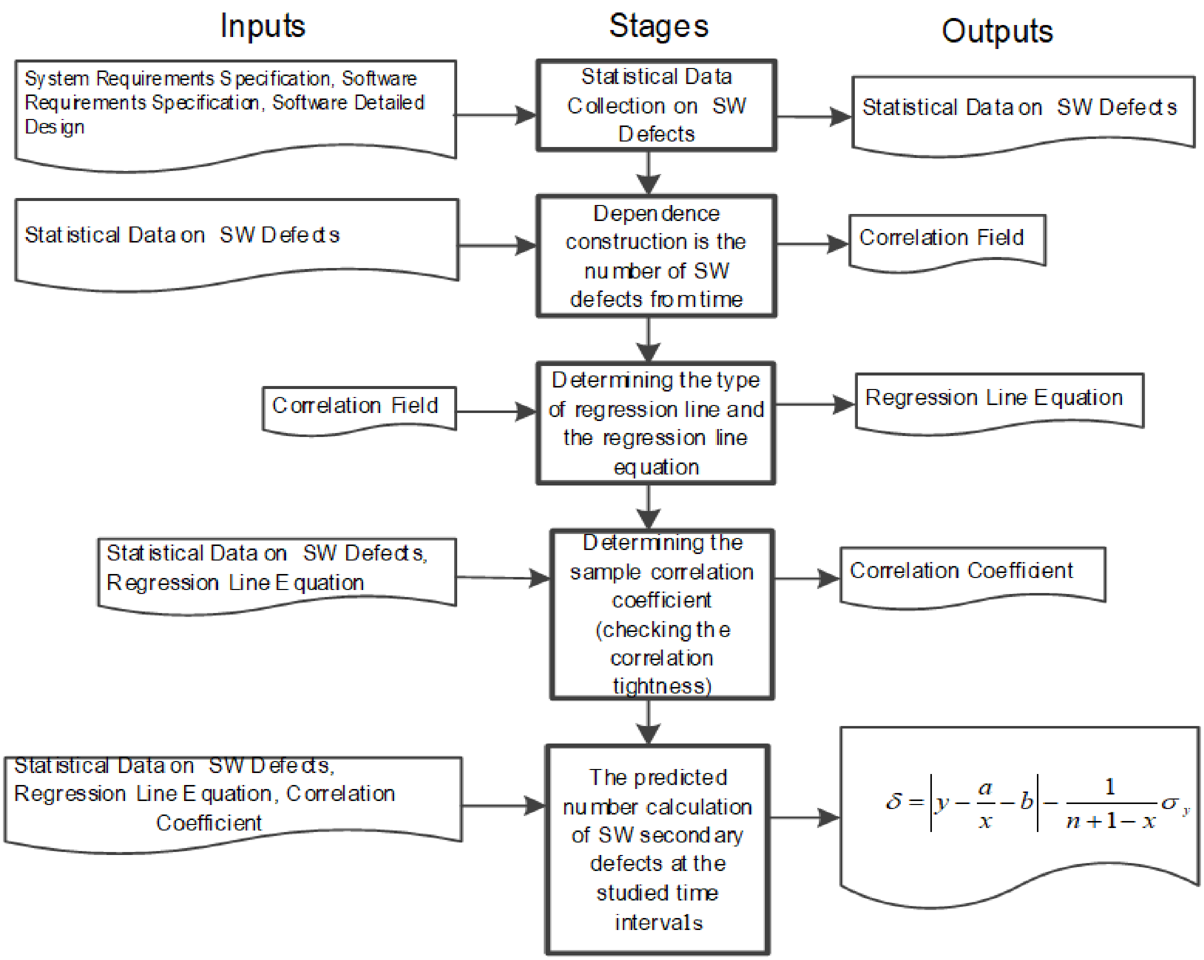
Inputs
Stages
Outputs
System Requirements Specification, Software Requirements Specification, Software Detailed Design
Statistical Data Collection on SW Defects
Statistical Data on SW Defects
Statistical Data on SW Defects
Dependence construction is the number of SW defects from time
Correlation Field
Correlation Field
Determining the type of regression line and the regression line equation
Regression Line Equation
Statistical Data on SW Defects, Regression Line Equation
Determining the sample correlation coefficient (checking the correlation tightness)
Correlation Coefficient
Statistical Data on SW Defects, Regression Line Equation, Correlation Coefficient
The predicted number calculation of SW secondary defects at the studied time intervals
$\delta = \left| y - \frac{a}{x} - b \right| - \frac{1}{n+1-x} \sigma_y$

Figure 1: Prediction of the number of secondary SW faults

4.2. Industrial case study: evaluation of the number of secondary SW faults in the Logic Module of RadICS platform

RadICS's (see Fig. 2) is the digital instrumentation and control platform used in safety and control systems applications in operating nuclear power plants [23]. It is a modular platform which includes a set of replaceable standardized modules such as logic module, digital and analog input/output modules. The functionality of each module is driven by the program logic implemented in the on-board FPGA(s). FPGA (Field-Programmable Gate Array) is an integrated circuit designed to be programmed through the use of hardware description languages, e.g. VHDL or Verilog [26, 27]. FPGA code is written in a description language, then is interpreted, synthesized, and ultimately produces hardware. As such, faults in the FPGA program code can introduce logic errors into the system.

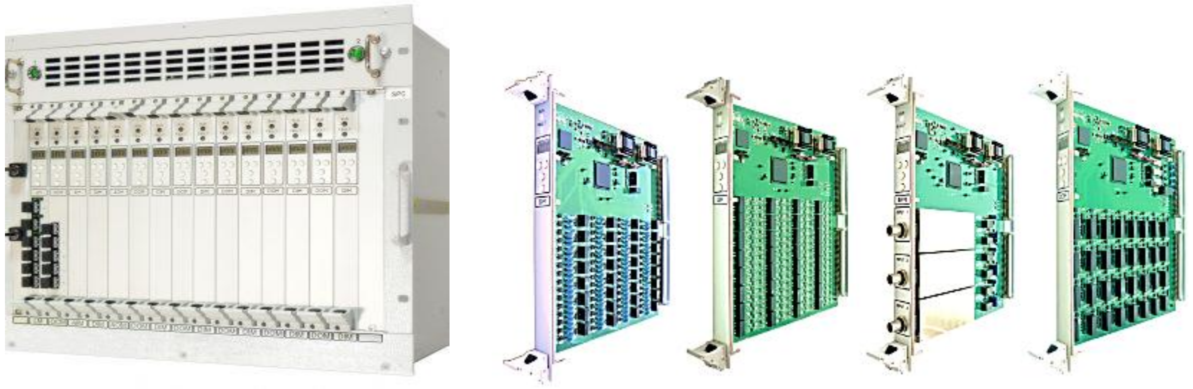

Figure 2: FPGA-based platform RadICS and its modules

In this section we report fault detection statistics (see Table 2) for RadICS's Logic Module collected during its functional testing and apply the discussed approach to predict the number of secondary faults. The Logic Module serves as the core of the entire RadICS platform. It implements the application logic and is used as a communication node linking other modules installed in the chassis together, performs self-diagnostics and controls communications with external chassis and systems.

Table 2
Statistics of the faults detected in RadICS's Logic Module during its functional testing

Month	1	2	3	4	5	6	7	8	9	10	11	12
Number of detected faults	14	12	8	7	7	6	5	6	3	2	1	1

Figure 4 depicts the fault detection statistics (NS_1) collected during functional testing and the power regression function (4) used for theoretical approximation of the testing results (NS_2). Values of the parameters of the regression function were estimated by the least-squares technique.

$$y = 12.66/x + 2.31 \tag{4}$$

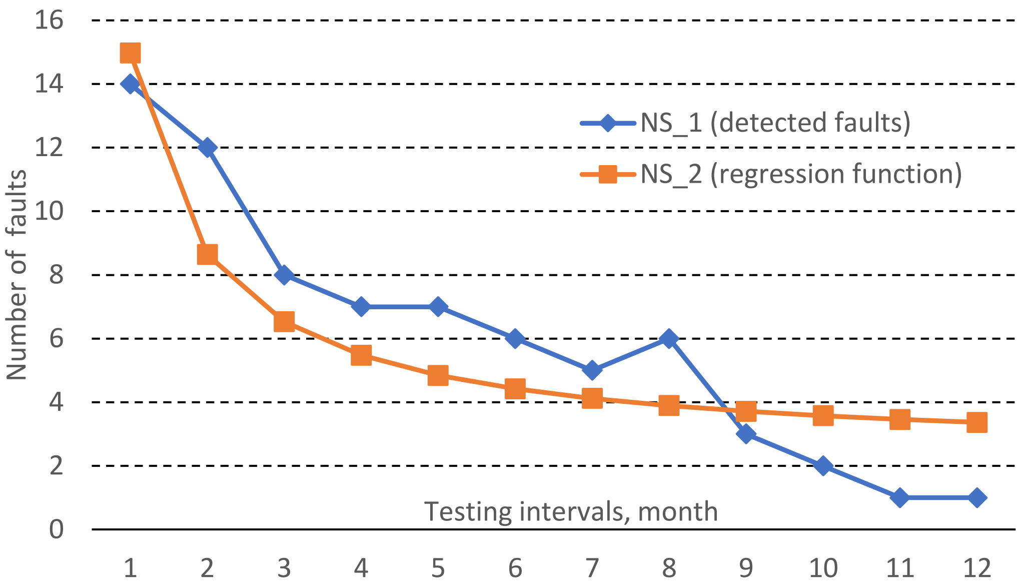

Figure 4: Fault detection statistics.

The number of secondary software faults n^{in} was calculated using (3) by pairwise comparison of NS_1 and NS_2 series. Obtained results are summarised in Table 3.

The number of predicted secondary faults can now be used to estimate the software failure rate λ_d with the help of software reliability growth models. Table 4 compares software failure rates predicted using Jelinski-Moranda SRGM with and without considering secondary software faults.

Table 3
The results of predicting the number of secondary software faults

Testing interval x	Number of detected faults y	$a+\frac{b}{x}$	$\left\lvert y-a-\frac{b}{x}\right\rvert$	$\left\lvert y-a-\frac{b}{x}\right\rvert-\frac{1}{13-x}\sigma_y$	Predicted number of secondary faults n^{in}
1	13	14,96916	1,969157	1,667813	2
2	11	8,639352	2,360648	2,031909	2
3	8	6,529417	1,470583	1,108970	1
4	7	5,474449	1,525551	1,123758	1
5	6	4,841469	1,158531	0,706515	1
6	5	4,419482	0,580518	0,063928	
7	4	4,118063	0,118063	–0,48463	
8	6	3,891998	2,108002	1,384776	1
9	3	3,716170	0,716170	–0,18786	
10	2	3,575508	1,575508	0,370132	
11	1	3,460421	2,460421	0,652356	
12	1	3,364514	2,364514	–1,25161	

Table 4
The results of software failure rate assessment λ_d

Testing interval x	Number of detected faults y	Predicted number of secondary faults n^{in}	Failure rate λ_d without considering secondary SW faults	Failure rate λ_d^* taking into account secondary SW faults
1	13	2	0,013580	0,013979
2	11	2	0,011183	0,011583
3	8	1	0,009286	0,009486
4	7	1	0,007788	0,007988
5	6	1	0,006490	0,006690
6	5		0,005392	0,005392
7	4		0,004493	0,004493
8	6	1	0,003495	0,003694
9	3		0,002596	0,002596
10	2		0,002097	0,002097
11	1		0,001797	0,001797
12	1		0,001598	0,001598
Average Software Failure Rate			0,005816	0,005949
Average decrease of the Software Failure Rate			0,001089	0,001126

Table 5 and Fig. 5 summarize cumulative numbers of both actually detected faults and predicted secondary faults. It is shown that the total number of faults detected during the testing period is equal to 67. The number of predicted secondary faults is 8 (approx. 12% of the detected faults).

Thus, one can assume that the potential number of software faults equals 75 which means that the software needs to be further tested to detect and remove the rest of the faults.

Table 5
Calculation of the relative error when taking into account secondary SW defects

Testing interval *x*	Number of detected faults *y*	Predicted number of secondary faults	Number of faults including secondary ones	Cumulative number of faults without secondary ones	Cumulative number of faults taking into account secondary ones	Predicted cumulative number of secondary faults	Relative deviation
1	13	2	15	13	15	2	0,133333
2	11	2	13	24	28	4	0,142857
3	8	1	9	32	37	5	0,135135
4	7	1	8	39	45	6	0,133333
5	6	1	7	45	52	7	0,134615
6	5		5	50	57	7	0,122807
7	4		4	54	61	7	0,114754
8	6	1	7	60	68	8	0,117647
9	3		3	63	71	8	0,112676
10	2		2	65	73	8	0,109589
11	1		1	66	74	8	0,108108
12	1		1	67	75	8	0,106667

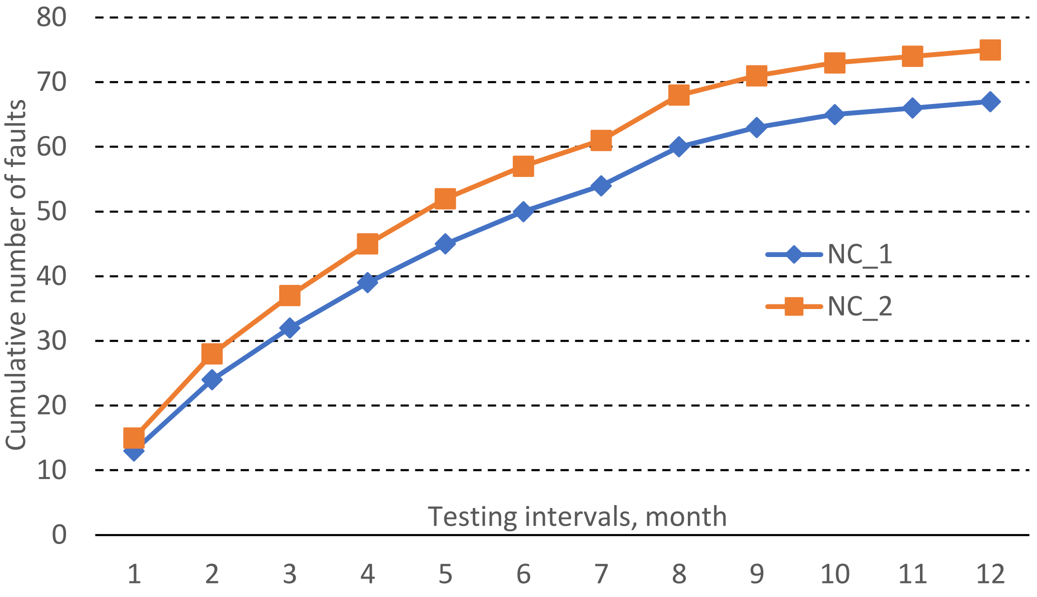

Figure 5: The cumulative number of software failures with (NC_2) and without (NC_1) accounting predicted secondary software faults

5. Conclusions

This paper discusses an importance of software reliability assessment. A great number of models have been proposed to predict software reliability (probability of failure, failure rate or mean time to failure) and the number of residual software faults.

Different models use different approaches to evaluate software reliability. For instance, one of the first attempt to predict the number of software faults was made by Maurice Halstead in 1977 in [23] where he proposed a number of program code's 'complexity' metrics used as predictors of program defects.

In this paper we consider a class of software reliability growth models which use fault detection statistics collected during software testing. These models put forward different assumption about statistical distribution of failures (e.g. time between failures follows the exponential distribution), number of faults (e.g. finite or infinite) and other limitations to simplify the process of software reliability assessment.

However, some of these assumptions might not be very realistic and, hence, affect accuracy of the reliability prediction. For example, many SRGMs assume that all detected faults are removed with certainty and no new faults are introduced in the software. However, our industrial experience and other studies show that the introduction of secondary faults is quite common and, hence, needs to be accounted during software reliability modelling.

In the paper we discuss different scenarios of introducing secondary faults and how to predict number of such faults. Finally, we discuss how different SRGMs like Jelinski-Moranda, Exponential, Schick-Wolverton, Musa and Lipov models can be modified to account secondary faults in order to improve accuracy of software reliability prediction. We use an industrial case study to demonstrate applicability of the proposed approach. Our results show that considering secondary faults helped to increase accuracy of software failure rate assessment up to 5%.

6. References

[1] IEC 61508:2010. Functional safety of electrical/electronic/programmable electronic safety-related systems, IEC Standards, 2010, 594 p.

[2] IEC 61513:2011. Nuclear power plants – Instrumentation and control important to safety – General requirements foe systems, IEC Standards, 2011, 210 p.

[3] O. Pomorova, O. Savenko, S. Lysenko, A. Kryshchuk, A.Nicheporuk. A Technique for detection of bots which are using polymorphic code. Communications in Computer and Information Science. 2014. Vol. 431. PP.265-276, ISSN: 1865-0929.

[4] L. Mirtskhulava, M. Khunjgurua, N. Lomineishvili, K. Bakuria, Software Reliability Prediction Model Analysis, International Journal of Computer, Information, Systems and Control Engineering, 2014, № 6, pp. 927-932. URL: https://publications.waset.org/9998645/software-reliability-prediction-model-analysis.

[5] I. Lakshmanana, S. Ramasamya, Selection of Right Software Reliability Growth Models for Every Software Project. International Journal of Control Theory and Applications. Volume 9. Number 40. 2016, pp.807-817. URL: https://serialsjournals.com/abstract/85226_89-cha-21.pdf.

[6] L. Xiaomei, X. Naiming, Grey-based approach for estimating software reliability under nonhomogeneous Poisson process. Jornal of Systems Engineering and Electronics. Volume 33, No. 2, April 2022, pp.360-369.

[7] H, Hecht, NASA-CR-145135: Measurement, estimation and prediction of software reliability, NASA, 1977.

[8] A.L. Goel, Software reliability models: Assumptions, Limitations and Applicability. IEEE Transactions on Software Engineering, Vol. SE-11, № 12, 1985, pp. 1411-1423. URL: https://dl.acm.org/doi/abs/10.1109/TSE.1985.232177.

[9] R.P. Garg, K. Sharma, R. Kumar, R.K. Garg, Performance Analysis of Software Reliability Models using Matrix Method. International Journal of Computer, Information, Systems and Control Engineering, 2010, № 11, pp. 31-38. URL: https://publications.waset.org/3581/pdf.

[10] ISO/IEC/IEEE 29148:2011 – Systems and software engineering – Life cycle processes – Requirements engineering, ISO, 2011.

[11] K. Wiegers, J. Beatty, Software Requirements, Microsoft Press, 2013.

[12] E. Hull, K. Jackson, J. Dick, Requirements Engineering, Springer, 2011.

[13] X. Li, Y. Yin, L. Fiondella, Y. Zhou, Software reliability analysis considering correlated component failures with coupling measurement framework. Journal of Systems

Engineering and Electronics. Vol. 26, № 5, 2015, pp. 1114-1126, doi: 10.1109/JSEE.2015.00121.

[14] Y. Shi, Infusing reliability techniques into software safety analysis. Proceedings of the Annual Reliability and Maintainability Symposium (RAMS 2015), pp. 1-5. doi: 10.1109/RAMS.2015.7105133.

[15] S.P. Chatzis, A.S. Andreou, Maximum Entropy Discrimination Poisson Regression for Software Reliability Modeling. IEEE Transactions on Neural Networks and Learning Systems. Vol. 26, № 11, 2015, pp. 2689-2701. doi: 10.1109/TNNLS.2015.2391171.

[16] A. Tickoo, P.K. Kapur, S.K. Khatri, Developing software reliability growth model for multi up gradations with faults of different severity and related release time problem. Proceedings of the 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), 2015, pp. 376-382. doi: 10.1109/ABLAZE.2015.7155023.

[17] P. Rotella, S. Chulani, D. Goyal, Predicting Software Field Reliability. Proceedings of the 2015 IEEE/ACM 2nd International Workshop on Software Engineering Research and Industrial Practice (SER IP), p.p. 62-65, DOI: 10.1109/SERIP.2015.20.

[18] S. Gnatiuk, Analytical model of reliability program of computer systems and software-controlled means of communication. Proceedings of the 13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET), № 16, 2016, pp. 90-92. doi: 10.1109/TCSET.2016.7451979.

[19] J. Zhang, Y. Lu, S. Yang, C. Xu, NHPP-based software reliability model considering testing effort and multivariate fault detection rate. Journal of Systems Engineering and Electronics. Vol. 27, № 1, 2016, pp. 260-270.

[20] H. Sukhwani, J. Alonso, K.S. Trivedi, I. Mcginnis, Software Reliability Analysis of NASA Space Flight Software: A Practical Experience. Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2016, pp. 386-397, doi: 10.1109/QRS.2016.50.

[21] A.N. Ivutin, E.V. Larkin, D.A. Perepelkin, Software errors and reliability of embedded software. Proceedings of the 2016 IEEE Conference on Quality Management, Transport and Information Security, Information Technologies (IT MQ IS)". № 23, 2016, pp. 69-71. doi: 10.1109/ITMQIS.2016.7751926.

[22] A. Choudhary, A.S. Baghel, O.P. Sangwan, Software reliability prediction modeling: A comparison of parametric and non-parametric modeling, Proceedings of the 2016 6th International Conference – Cloud System and Big Data Engineering (Confluence), № 19, 2016, pp. 649-653. doi: 10.1109/CONFLUENCE.2016.7508198.

[23] Radiy. Products for NPP. URL: http://radiy.com/ru/produktsiya-dlya-aes/produktsiya/platforma-radics.html.

[24] V.S. Kharchenko, O.M. Tarasyuk, V.V. Sklyar, V.Yu. Dubnitsky. The method of software reliability growth models choice using assumptions matrix, Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC’02), 2002, pp. 541-546.

[25] M.H. Halstead, Elements of Software Science, Amsterdam: Elsevier Science Ltd., 1977.

[26] M. Rafiquzzaman, Steven A. Mcninch, Digital Logic: With an Introduction to Verilog and FPGA-based Design, Wiley, 2019.

[27] S. Naumenko, V. Moskalets, O. Odarushchenko, E. Odarushchenko, V. Peschanenko, L. Degtyareva, O. Letychevskyi, Formal methods of FPGA Project Verification Flow, Proceedings of the 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, Cracow, Poland, 2021, pp.1141-1146.