Design, Evaluation and Implementation of Algorithms for Predicting Software Defects and Failures During the Operational Phase

Kazuhira Okumoto
Sakura Software Solutions, LLC
Naperville, IL USA
Email: kokumoto75@gmail.com

Abstract – In a world where AI and automation are prevalent, ensuring software and hardware reliability is crucial. While defect modeling during the coding and testing phases is well-studied, there remains a gap for both theoretic and/or practical models for the operational phase. This phase, starting post-deployment (when software is in use by customers), is critical as defects can cause customer-impacting system downtime. This paper integrates approaches from software and hardware reliability modelling to propose methods for predicting operational phase defects and failures. For this phase, we propose to approximate the software failure rate as a constant and develop a digital tool for system reliability modeling, that correlate software reliability with hardware profiles for comprehensive assessments. While software and hardware reliability modelling have both been separately well-studied, this paper is a ground-breaking attempt to combine them in the context of a deployed software solution.

Index Terms – *Customer defects, software failures, operational software reliability, piecewise modelling, digital engineering*

I. Introduction

As the world becomes more AI-powered, automated, and robotic, software and hardware reliability is crucial. Usually, such reliability depends on processes that use predictive models to detect defects in software or hardware. Software defects, also called faults or bugs, are mistakes or glitches in a computer program or system, causing wrong or unexpected results, which affect software reliability greatly.

While extensive research has focused on modeling and analyzing defects during the coding and testing phases of software development, a notable gap exists in practical models dedicated to the operational phase. This operational phase commences upon software deployment at customer sites, where defects hold the highest criticality due to their potential to cause system downtime. This contrasts sharply with the well-studied operational phase of hardware reliability. Hence, integrating software and hardware reliability, despite their distinct modeling approaches, is imperative given their interconnected impact on system failures. Furthermore, the necessity for combined reliability evaluation solutions is underscored by the need to facilitate architectural comparisons during the pre-construction design phases.

A software release comprises newly developed features or functionalities, which undergo a lifecycle development process involving requirements specification, software design, coding, and testing against predefined criteria. This software lifecycle is depicted in Figure 1. Once the testing phase concludes, the software undergoes acceptance testing at customer sites before being deployed for commercial operation. During this process, software defects are identified by both internal testers and customers. Software quality improves iteratively through a find-fix process, encompassing internal and customer tests and operations. This often leads to the development team addressing fixes for both internal test defects and customer-reported defects post-initial delivery to the customer site. However, defects continue to surface during the operational phase after commercial

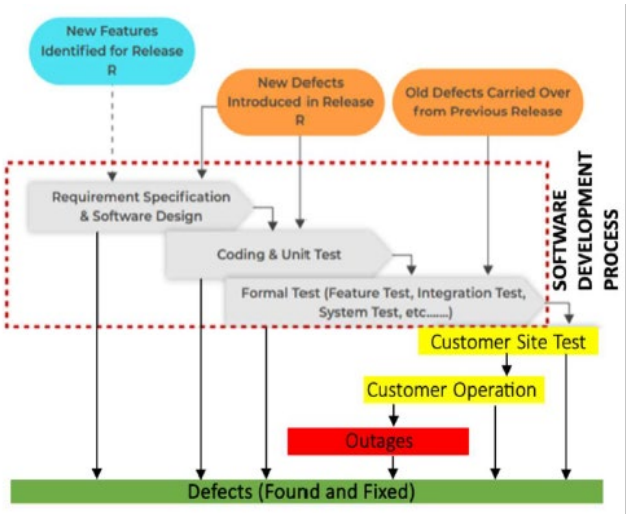

Figure 1. A sample software life cycle with defect

deployment, some of which may result in system outages or failures in the field. Understanding the distinction between defects and failures forms a crucial foundation for the proposed project. We assume critical severity defects to represent software failures based on the severity classification of software defects.

In practice, two main scenarios arise: A) Software development is conducted in-house, and B) Software development is outsourced to a subcontractor, where system testing is performed internally before delivering the final software product for commercial use. In Case A, customer defect data from system tests and operations are provided by the customer. In Case B, such data are available in-house, though defect data during development and testing stages may only be accessible if provided by the subcontractor. For example, in the telecommunication industry, equipment suppliers such as Nokia and Ericsson handle software development, while service providers like AT&T, Verizon, and T-Mobile perform system tests and deploy the software product in a nationwide network to provide wireless services to end-users like cell phones and personal computers. The work in this paper will be particularly beneficial to service providers.

Most software reliability growth models that exist today [1] – [7] mainly address Case A, where they use defects discovered during the testing phase to generate defect prediction curves like exponential or S-shaped. However, a new cloud-based tool [9] – [15] has been recently created, using SaaS technologies [8], that incorporates defect close and open curves. A new modeling approach has also been proposed, that combines a series of piecewise exponential models and automated prediction algorithms. There is not much research on Case B [16] – [17]. This paper concentrates on Case B and outlines a plan for developing a digital tool for assessing software and hardware system reliability [18] – [19].

Figure 2 illustrates typical defect trends, including release level (in blue), internal (in yellow), and customer defects (in red). The internal defects are typically found during the formal test phase. The internal defect curve generally slows down after the last delivery date (D2), and the customer defect curve starts at the first delivery date (D1). We also show a defect prediction curve (in dashed black) generated at the last delivery date with our online tool, STAR [14] – [15]. The prediction curve is generally higher than the actual defect curve because the prediction curve assumes the intensity of the internal test continues. The gap between these curves

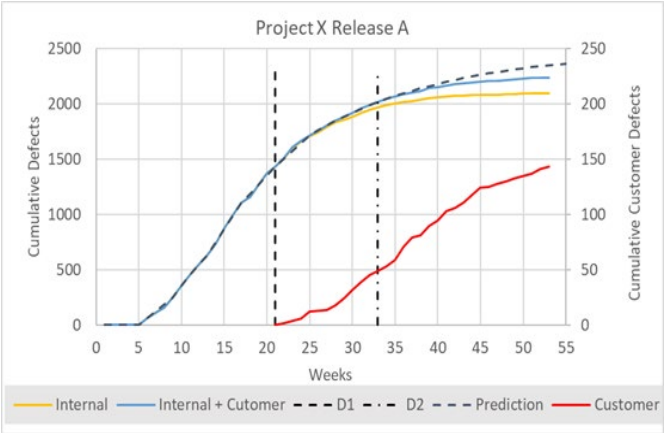

Figure 2. Cumulative view of internal and customer injection and find-fix processes defect curves with a predicted curve at delivery (D2)

represents defects not found in this release and will be carried over to the next release. Also shown in Figure 2 is the customer defect curve, which is the difference between the release-level and internal defect curves.

In this paper, we focus on customer defects and failures, as highlighted in the yellow boxes in Figure 1. A software failure denotes a system outage caused by a software defect during an operation period. Software reliability denotes the probability of the system operating without a software failure during a specified period, typically measured in failures per year. Software availability represents the likelihood of the software system being operational. Both software reliability and availability metrics are defined for an operation period. Despite frequent references to software reliability in literature, this field requires more attention. There is often an assumption that software defects equate to software failures, and the software defect curve during the operation phase will follow the extension of the defect prediction curve.

The rest of this paper is organized as follows: Sections II and III will present innovative approaches for automatically predicting customer defects and failures during the operational phase. A key aspect of our paper will be to estimate the software failure rate as constant throughout the operation period, which forms a foundational element of our research. Detailed technical discussions regarding these methodologies will be provided in these sections. Additionally, in Section VI, we will utilize multiple real-world datasets to illustrate how defects and failures can be accurately represented as piecewise straight-line curves.

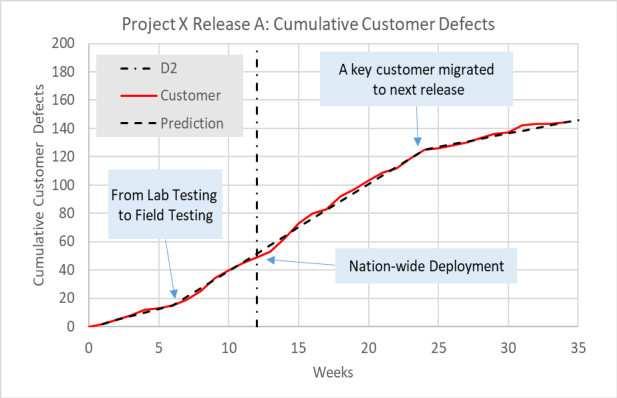

Figure 3. Cumulative view of customer defects

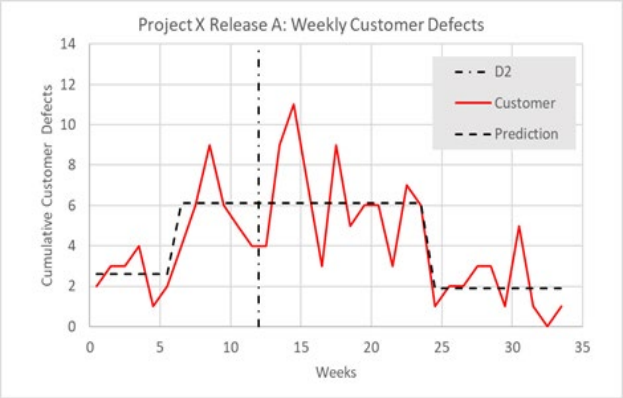

Figure 4. Weekly view of customer defects with prediction

Section V will describe a plan for creating a digital tool for self-service system (software and hardware) reliability modeling. Our main goal is to develop a comprehensive system (software and hardware) reliability model, that includes algorithmic analytics automation and the creation of a digital engineering tool for holistic system-level reliability evaluations. First steps involve linking common software reliability growth models with the hardware bathtub profile. Highlighting the constant defect growth rate during the customer operational phase matches the flat phase seen in hardware curves. By using the software failure rate based on failure intensity from observed failures during customer software operation, we aim to achieve this link effectively. Next, integrated software and hardware models will be suggested, treating software similar to hardware. This approach will determine system reliability through reliability block diagrams (RBDs), providing metrics such as operational software reliability and availability.

II. Prediction Model for Software Defects During the Operational Phase

Different stages of software usage in the operational phase have different rates of finding defects: early customer testing, customer testing, post-deployment, and customer migration to newer releases. To handle this variation, we want to create a defect trend model that covers the whole operational phase. This model will be shown as a series of straight lines based on empirical data. Figure 3 shows this idea with sequential lines. For more detail, Figure 4 gives a weekly report of the trend.

Our initial studies in this direction suggest a segmented linear equation is expressed in a mathematical form as:

$$m(x) = m(x_{j-1}) + \theta_j (x - x_{j-1}) \quad \text{for } x_{j-1} \le x \le x_j \qquad (1)$$

where $m(x)$ represents the cumulative defects at time x, and x_j represents the j-th breakpoint. The parameter θ_j is the slope of the j-th straight line, representing the defect rate.

The main technical challenge in reaching our goal is to develop an algorithm that can automatically detect transition points (or change points) between each straight line, and adjust to the different patterns in the data. It's important that this algorithm works smoothly in real-time, and considers new defect data as it comes in. This will result in a segmented linear representation of software defects during the operational phase. This algorithm can also be applied to hardware failure data, not just software. Moreover, we will compare the actual hardware failure rate with the rate estimated from the bill of materials (BOM) to provide comprehensive insights.

We have developed a novel method for automatically detecting transition points where the slope of the straight line changes significantly. First, we find these transition points or time intervals where the defect trend shows a noticeable shift, as shown in Figure 3. Employing a straightforward straight-line model, we continuously track the slope and detect significant changes in the relative change of consecutive slope values. An inflection point is established when this relative change surpasses a predetermined threshold.

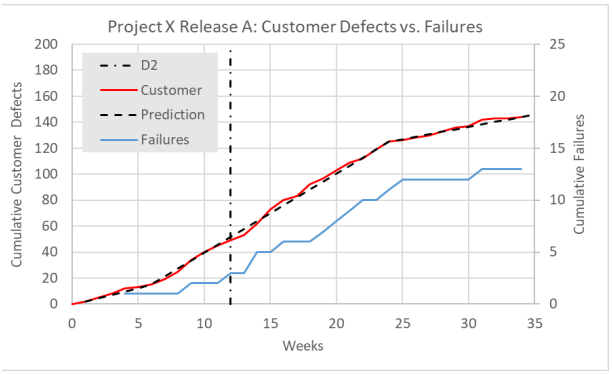

Figure 5. Cumulative view of customer defect

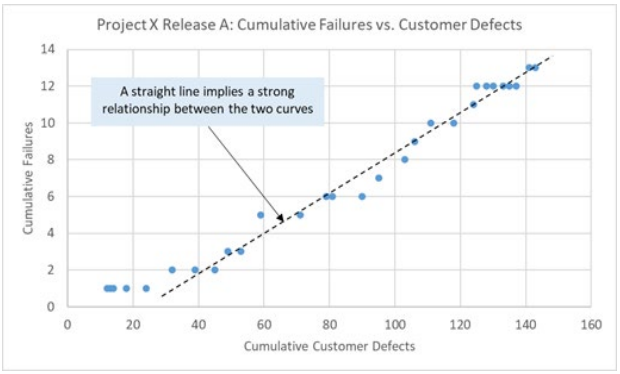

Figure 6. A plot of cumulative failure data vs. and software failure curves cumulative customer defect data

Upon locating the transition point, we adjust the start date to correspond to this point and repeat these steps until reaching the end of the defect data. This iterative process yields the final predicted curve, depicted in Figure 3, comprising a series of piecewise straight-line models, as detailed in equation (1).

Our algorithm seamlessly provides numerical solutions in real time as new defect data emerges. By amalgamating best-fitted straight lines for individual time segments between transition points, we construct a piecewise straight line. Utilizing the Project X Release A data, we illustrate the resulting defect prediction curve in Figure 3, aligning with the expected outcomes outlined previously. Further examination of the algorithm's robustness will be expounded upon in Section 4. Additionally, a weekly perspective of actual defects alongside the predicted curve is presented in Figure 4.

III. Prediction model for software failure rates

The occurrence of software failures typically remains relatively infrequent compared to customer defects, posing a statistical challenge in data analysis. To overcome this hurdle, we propose the implementation of a transformation function aimed at converting the customer defect curve into a corresponding software failure curve. Both curves are visually represented in Figure 5. Initial investigations indicate that the quantile-quantile (Q-Q) plot technique holds promise in facilitating this transformation process. The rationale behind the Q-Q plot technique is demonstrated through a graphical representation of cumulative failures plotted against cumulative customer defects, as illustrated in Figure 6. This graphical representation unveils a linear alignment, suggesting a significant correlation between the curves.

This involves transforming the original time variable x into a new time scale u. To achieve this, we must determine an appropriate transformation function f that preserves the segmented linear configuration. The resulting software failure curve will thus be represented as follows:

$$n(u) = n(u_{j-1}) + \lambda_j (u - u_{j-1}) \quad \text{for } u_{j-1} \le u \le u_j \qquad (3)$$

The software failure rate can be obtained from the value of λ_j right after the software deployment date.

The failure rate curve $n(u)$ can then be derived from the defect curve $m(x)$ via a transformation function f as:

$$n(u) = f(m(x)) \qquad (4)$$

It's crucial to acknowledge that conventional statistical techniques like correlation coefficients are impractical due to the dynamic nature of defect and failure data. This dynamism presents a technical obstacle in data management. Our proposed approach entails horizontal and vertical shifts to align the customer defect curve with the software failure curve. We aim to develop an algorithm capable of numerically solving a non-linear optimization problem, thereby pinpointing the optimal-fit software failure curve using the customer defect curve as a guiding metric. This algorithm will automatically generate the software failure curve, depicted as a series of interconnected straight lines in a piecewise manner.

The statistical transformation function comprises two components: horizontal and vertical shifts. Put

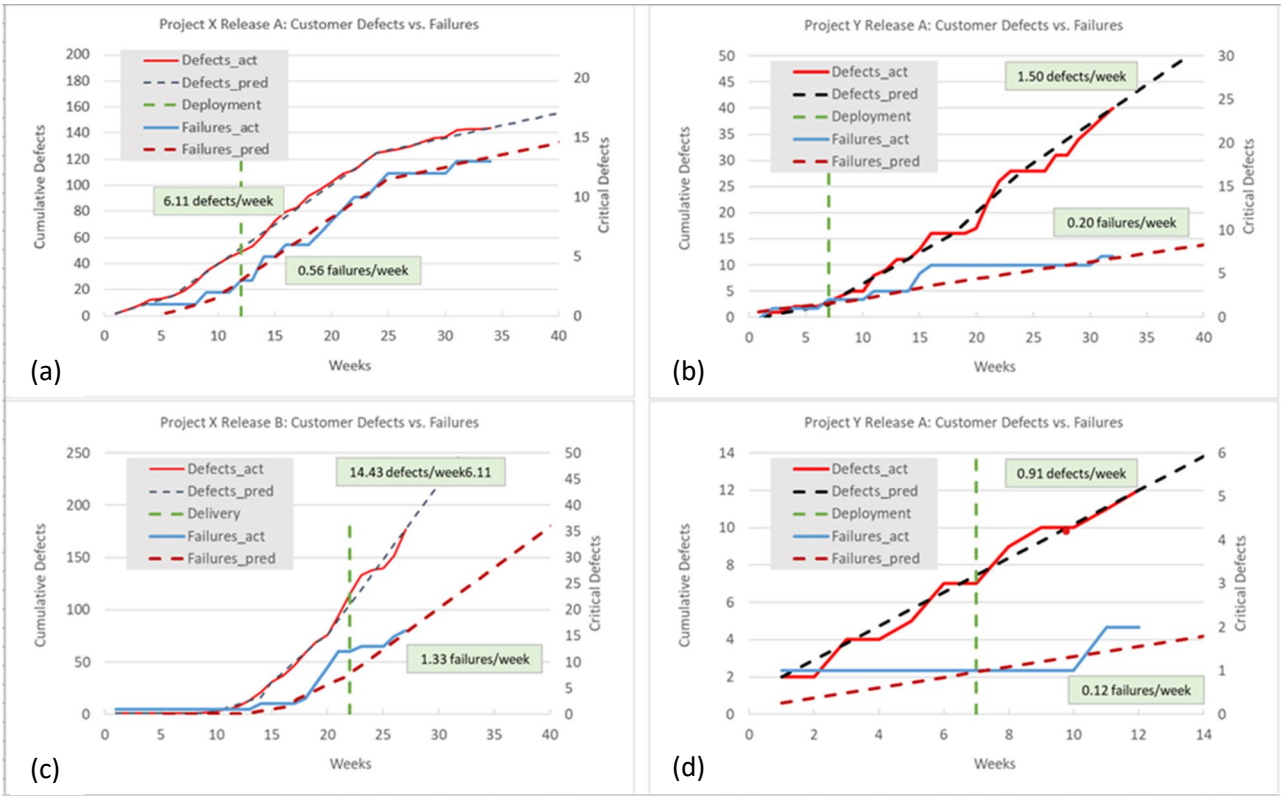

Figure 7. Prediction results of software customer defects and failures for four project data

differently, the initial curve, denoted as the prediction curve in dashed black, undergoes a transformation involving horizontal and vertical shifts to closely align with the actual failure curve. The transformation function, mapping (x, m(x)) coordinates to (u, n(u)), is detailed in equations (5) and (6) for horizontal and vertical shifts, respectively.

$$u = \alpha + \beta\, x \qquad (5)$$

$$n(u) = \gamma\, m(x) \qquad (6)$$

The parameter α signifies a fixed delay in weeks from the defect curve to the failure curve, while β represents an additional delay in the defect curve. The parameter γ is determined as the ratio of defects to failures, indicating defects per failure. That is, this ratio represents the portion of defects resulting in a failure.

This problem entails non-linear optimization with three variables (or parameters) and aims to minimize the total distance between the predicted defect curve and the actual failure curve. Solving this problem involves iterative numerical analysis with three-level loops and multiple iterations for each level. Initial values, increments, and maximum iteration counts must be defined for each level. For a detailed explanation of this numerical analysis, please refer to [14].

Figure 7(a) depicts the final predicted failure curve (shown as dashed red), which should closely align with the actual failure curve (depicted in blue). To validate the predicted failure curve, actual failure data can be overlaid, as demonstrated in Figure 7(a). The figure also displays the defect rate and failure rate as 6.11 defects/week and 0.56 failures/week, respectively, corresponding to the slopes of the two predicted curves at deployment (D2). The ratio of defects to failures, calculated as 0.092, represents the value of γ, indicating that only 9.2% of defects will cause a software failure.

In practice, the number of failures is typically limited, posing challenges for the application of statistical methods. The transformation function method outlined in this section effectively addresses this challenge. In the subsequent section, we will utilize Project X Release B data to exemplify the application of the transformation method for a limited dataset.

IV. Application of the prediction methods to other project data

To assess the effectiveness of the proposed models, we will utilize four distinct real-world defect datasets obtained from significant software development projects within the telecommunications sector. These datasets comprise customer defects identified during both internal and operational phases, categorized based on severity levels.

In practice, there are primarily two scenarios: A) Software development conducted in-house, where the software product is delivered directly to a service provider and B) Software development outsourced to a subcontractor, with a service provider conducting the system test internally. In Case A, customer defect data from system tests and operations are provided by the service provider. In Case B, while customer defect data are available internally, defect data during the development and test stages are only accessible if provided by the subcontractor.

A. In-house software development perspective

We have two consecutive releases, A and B, for each of the two projects, X and Y, at our disposal to validate the prediction models outlined in Sections II and III. Both projects constitute integral elements of a telecommunications network system known as a base transceiver station (BTS), which facilitates wireless communication between user equipment (e.g., a cell phone) and a network. Project X involves a highly demanded large-scale software development for 5G, while Project Y encompasses a stable mid-size software development for 4G. Figure 7 provides an overview of customer defects and failures, along with predictions.

In both Projects X and Y, Release A boasts a significant number of defects, making it suitable for prediction methods. However, Release B lacks sufficient failure data for both projects. To address this, we utilized the values of α, β, and γ of the transformation function obtained from Release A to predict failures for Release B in both projects. This approach is deemed reasonable for both cases. We have demonstrated that both the defect rate and the failure rate remain constant after deployment. Moreover, we can reasonably employ the transformation function from the previous release to predict the failure rate if the failure data is insufficient.

B. Software development by a subcontractor

In this scenario, we (acting as the customer) subcontract the software development, conduct the system test, and provide the service to the end-user upon readiness. We possess defect data from both the system test and the operation phases. The dataset below depicts defects identified during the system test after several deliveries of a software product by a subcontractor. Unfortunately, detailed information regarding the project's background is limited.

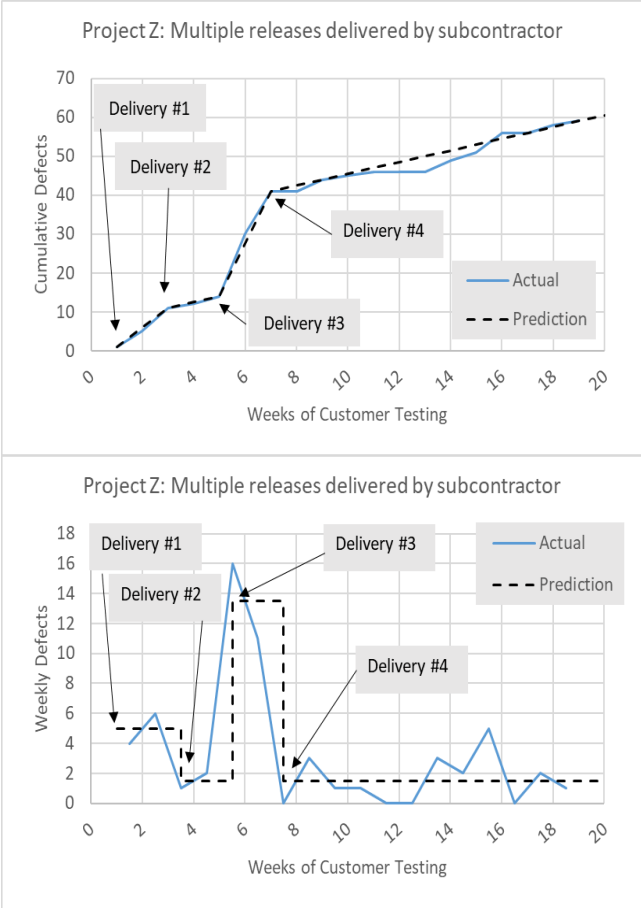

Figure 8. Customer test defects with multiple deliveries by a software development subcontractor

Figure 8 showcases the cumulative defects detected during the system test. Over the course of the period, the subcontractor made four deliveries. Notably, the initial delivery (Delivery #1) appears to encompass a relatively small portion of the total features, followed by a maintenance load (Delivery #2) featuring bug fixes. Subsequently, a third delivery (Delivery #3) introduced most of the remaining features, succeeded by another maintenance load (Delivery #4) with bug fixes. We have observed a similar result from our NASA data. It is evident that the final maintenance load potentially contained defects while maintaining the system's stability. Once again, we have illustrated that customer defects can be elucidated using piece-wise straight lines.

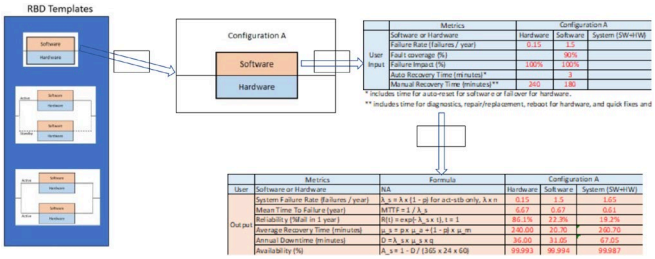

Figure 9: System Reliability Block Diagram. Sample graphical editor for a single-unit system (software + hardware)

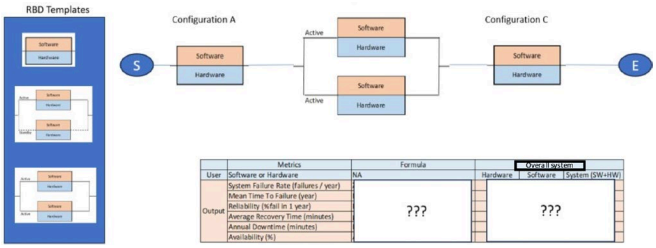

Figure 10: Reliability Block Diagram. Sample graphical editor for system reliability block diagram. A combination of two single-unit configura6ons and one two-unit active-active configuration.

V. Self-service system for system reliability engineering

This section outlines a plan for deploying a self-service system dedicated to system reliability engineering encompassing both software and hardware aspects. As the project is currently underway, we will provide a high-level overview of our digital tool.

A. Models for Integrated Software and Hardware Failure

The proposed system will utilize the algorithms and models developed in Sections II and III and use statistical methods to correlate the failure model developed above with the flat phase of the bathtub hardware curve to create a combined software and hardware failure model.

B. System Reliability Modeling Digital Tool

The research endeavors to design and develop a digital application aimed at facilitating the interactive creation of system reliability block diagrams (RBDs), integrating both software and hardware components. The system will offer users several preconfigured RBD templates to select from, including a single-unit system, a two-unit standby system, and a two-unit active-active system, as depicted in Figure 9.

Users will have the capability to extract hardware failure data from a widely utilized failure rate database, leveraging Bills of Material (BOM). Furthermore, the tool will integrate formulas for each RBD template to compute crucial system reliability metrics such as Mean Time Between Failures (MTBF) and availability.

Additionally, the application will empower users to merge these templates, enabling the creation of a comprehensive representation of the entire system. Users will be able to visualize software failure rate predictions alongside the RBDs, facilitating a more thorough comprehension of the system's overall reliability.

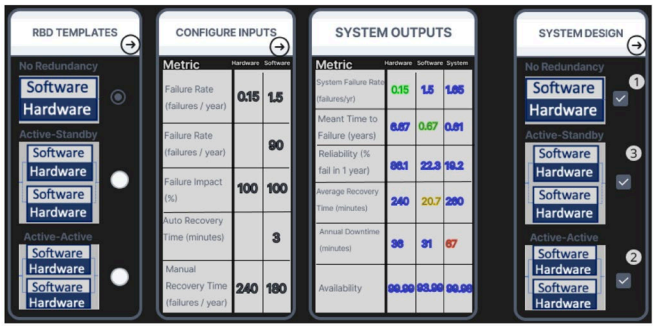

Figure 11: Potential User Journey While Using the Reliability Block Diagram Configurations

Figure 10 illustrates the graphical editor combining two single-unit configurations and one two-unit active-active configuration, while Figure 11 showcases a potential user interface journey elucidating how the reliability block diagram configurations may be implemented. In this illustration, the initial screen depicts a user selecting one of the available configurations and proceeding by clicking the next arrow. Subsequently, the user is directed to a screen where certain inputs can be configured, and upon completion, the system generates output metrics associated with the configuration and inputs. The final screen demonstrates the possibility of selecting multiple RBD templates and specifying the order in which the selected templates should be arranged. It's important to note that this is merely a mockup, and the actual designs and implementation are expected to occur during the implementation phase.

VI. Conclusion

In this paper, we have demonstrated that customer defects and software failures during the operational phase can be effectively modeled using a piecewise straight-line approach. We have presented novel algorithms to automate the prediction process, and their reliability has been verified by analyzing several real-world project datasets. This finding, implying a constant rate of operational software reliability, opens the

possibility of developing a digital tool for self-service system reliability modeling, covering both software and hardware components. We have described our initial design plan for the online tool, laying the foundation for further development and implementation.

Acknowledgment

The author would like to thank Dr. Rashid Mijumbi, Rory Harpur, Joseph Good, Mike Okumoto, and Andrew Raposo for their contributions to STAR and to this paper.

References

[1] H. Pham, Software Reliability. Berlin, Heidelberg: Springer-Verlag, 1999.

[2] J. D. Musa. A theory of software reliability and its application. IEEE Transactions on Software Engineering, SE-1(3):312-327, 1975.

[3] H. Pham and X. Zhang. NHPP software reliability and cost models with testing coverage. European Journal of Operational Research, 145(2), 2003.

[4] D. D. Hanagal and N. N Bhalerao. Literature Survey in Software Reliability Growth Models. In Software Reliability Growth Models, pages 13-26. Springer Singapore, Singapore, 2021.

[5] P K Kapur, S. Bhushan, and S. Younes. An exponential SRGM with a bound on the number of failures. Microelectronics Reliability, 33(9):1245-1249, 1993.

[6] S. Yamada, M. Ohba, and S. Osaki. S-Shaped Reliability Growth Modeling for Software Error Detection. IEEE Transactions on Reliability, R-32(5):475-484, 1983.

[7] A. L. Goel and K. Okumoto. Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures. IEEE Transactions on Reliability, R-28(3), 1979.

[8] M. Dan. The business model of "Software-As-A-Service". In Proceed-ings - 2007 IEEE International Conference on Services Computing, SCC 2007, 2007.

[9] K. Okumoto, A. Asthana, and R. Mijumbi. BRACE: Cloud-based software reliability assurance. In Proceedings - 2017 IEEE 28th International Symposium on Software Reliability Engineering Workshops, ISSREW 2017, 2017.

[10] R. Mijumbi, K. Okumoto, A. Asthana, and J. Meekel. Recent Advances in Software Reliability Assurance. In 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pages 77-82, 2018.

[11] R. Mijumbi, K. Okumoto, and Abhaya Asthana. Towards automated, end-to-end software defect prediction. In Proceedings - 25th ISSAT International Conference on Reliability and Quality in Design, 2019.

[12] R. Mijumbi, K. Okumoto, and A. Asthana. Software Reliability Assurance in Practice. In Wiley Encyclopedia of Electrical and Electronics Engineering. 2019.

[13] K. Okumoto, R. Mijumbi, and Abhaya Asthana. Software Qual-ity Assurance. In Moharmnad Abdul Matin, editor, Telecommunication Networks, chapter 3. IntechOpen, Rijeka, 2018.

[14] K. Okumoto. Early Software Defect Prediction: Right-Shifting Software Effort Data into a Defect Curve. In 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pages 43-48, 2022.

[15] K. Okumoto, “Digital Engineering-driven Software Quality Assurance-as-a-Service,” in Proceedings - 29th ISSAT International Conference on Reliability and Quality in Design, 2023.

[16] X. Zhang and H. Pham. Software field failure rate prediction before software deployment. Journal of Systems and Software, 79(3), 2006.

[17] K. Okumoto, "Experience Report: Practical Software Availability Prediction in Telecommunication Industry," in Proceedings - International Symposium on Software Reliability Engineering, ISSRE, 2016. doi: 10.1109/ISSRE.2016.19.

[18] M. Zhu and H. Pham, "A novel system reliability modeling of hardware, software, and interactions of hardware and software," Mathematics, vol. 7, no. 11, Nov. 2019, doi: 10.3390/math7111049.

[19] Relyence, "A Deep Dive into System Modeling Using Reliability Block Diagram (RBD) Analysis," https://relyence.com/wp-content/uploads/2021/09/A-Deep-Dive-into-System-Modeling-using-Reliability-Block-Diagram-RBD-Analysis.pdf, 2021.