Chapter 3

Software Quality Assurance

Kazu Okumoto, Rashid Mijumbi and Abhaya Asthana

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.79839

Abstract

Telecom networks are composed of very complex software-controlled systems. In recent years, business and technology needs are pushing vendors towards service agility where they must continuously develop, deliver, and improve such software over very short cycles. Moreover, being critical infrastructure, Telecom systems must meet important operational, legal, and regulatory requirements in terms of quality and performance to avoid outages. To ensure high quality software, processes and models must be put in place to enable quick and easy decision making across the development cycle. In this chapter, we will discuss the background and recent trends in software quality assurance. We will then introduce BRACE: a cloud-based, fully-automated tool for software defect prediction, reliability and availability modeling and analytics. In particular, we will discuss a novel Software Reliability Growth Modeling (SRGM) algorithm that is the core of BRACE. The algorithm provides defect prediction for both early and late stages of the software development cycle. To illustrate and validate the tool and algorithm, we also discuss key use cases, including actual defect and outage data from two large-scale software development projects from telecom products. BRACE is being successfully used by global teams of various large-scale software development projects.

Keywords: software quality & reliability assurance, telecommunication software, software defect measurements & modeling

1. Introduction

Emerging technology breakthroughs in a number of fields, such The Internet of Things, 5G, artificial intelligence, etc., are propelling a continuous increase in the size and complexity of software in communication systems. In such applications, software must have—among others—two important characteristics: (1) quality and (2) agility. Software quality is important to

limit downtime and vulnerabilities that might have unprecedented consequences for the anticipated societal-critical applications, while agility helps to continuously improve services to meet customer needs. In order to meet business objectives, and supported by advances in virtualization technologies, Telecom service providers are recently moving towards service agility, where the aim is to create new or improve the services provided to their users [1]. To this end, network software vendors must have the capability to continuously deliver and integrate software from which such services are composed.

These requirements mean that development teams must be able to produce quality software in very short time periods. This calls for accurate ways of planning team resource requirements or software size (number/complexity of features) ahead of time to avoid failing to meet customer needs. A useful approach for this assessment is *predicting the number of defects* during the requirements and design phase, which may allow for time to take preventive actions such as additional reviews and more extensive testing, finally improving software process control and achieving high software quality [2].

1.1. Software development phases

Software quality can be significantly affected by the number of defects and amount/type of testing carried out at the different stages of the software development phase. In this subsection, we examine the defect flow through various development phases and in the field. Once a set of new features is identified, each of them goes through the phases shown in **Figure 1**. It can be observed that there are a number of activities overlapping in time (*x-axis*). New defects are introduced during the requirement, design and coding phases while old defects are carried over from previous releases into the current release. The thickness of the arrows indicates the relative numbers of defects expected at each point in time. Defects are found and removed or

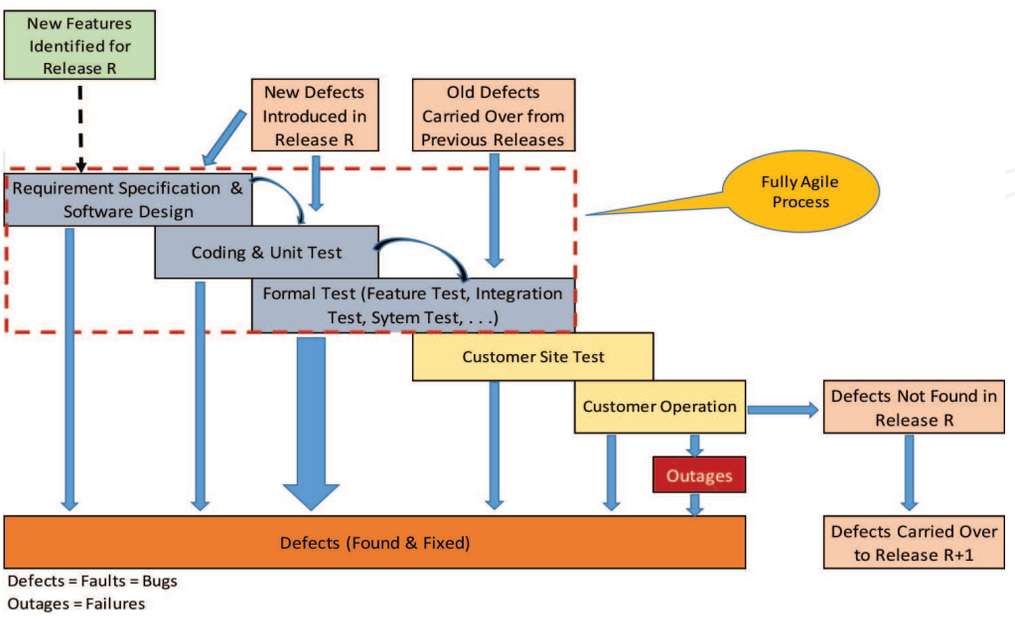

Figure 1. Defect flow development, test, field (traditional vs. agile).

fixed during every development phase as well as during customer site test and actual operation periods. In a traditional development process, most defects are found by testers, independent of developers. The test phase includes integration, feature, system, robustness, stability, and performance testing, just to name a few.[1] Most software defects are expected to be found and removed during these testing phases.

A proportion of defects that remain at the end of the test phase are normally encountered during operation in the field, and usually, only a fraction of them lead to failures or outages. However, an outage puts a system down, making it go through a process of respond-recover-resolve. Finally, some of the residual defects will usually not be found even during field operation. These then become base or old defects for the following release.

1.2. SRGM background

The quality of software is measured in terms of software defects found during the customer operation period. SRGM is usually used to quantify the quality of software products before they are delivered to customers. SRGM is a very well-studied subject, with over 200 SRGMs developed since early 1970s [3–15]. These models can generally be categorized as either being *parametric* or *non-parametric*.

Parametric models are based on explicit mathematical expressions and physical interpretation. They were originally designed for system test but continue to be extended to include functional, integration, and other earlier test phases. They are the earliest developed models, are easily understood, and widely used. Most of the frequently used parametric models can be systematically grouped based on the shape of the function (i.e. *S-shaped* curve or *exponential* curve). S-shaped curve models use various types of S-shaped distributions such as Gamma, logistic, Weibull functions to match with actual data trends [6, 9], while exponential models (e.g. [3, 4, 7, 10, 11]) consider that defects are found at a constant rate through the different test phases. S-curve models have flexibility in describing different shapes of the trend since they have more than two parameters. Compared with S-shaped models, exponential models are simple with only two parameters and have been successfully demonstrated with practical data from large-scale software development projects [16, 17]. There have been many comparison studies performed and various tools have been developed for evaluating individual models in terms of how well each model fits to the data [18, 19].

Recently, non-parametric SRGMs are being proposed. Such models typically use *machine learning* techniques considering the inherent characteristics of the software (such as code size, test resources, test duration, complexity, etc.) as input. Although non-parametric models are promising, their use in practical applications is still limited by their need for (big amounts of) input data which is not always available, or whose quality may be unacceptable. Therefore, practical implementations and application of SRGMs are still mainly based on parametric models. Even then, there is no single model which can be used in every situation [20]. Models that work well during the early development phases are typically not efficient in the later

[1] It is worth noting that test phase names vary across projects and organizations.

stages, in which case a complete solution may require more than one model. In addition, the fact that the rate at which defects are found can change overtime requires that even for the same phase, the same model may need to be applied more than once each covering a specific trend. In practice, applying such models in an efficient way can be effort intensive and usually requires an expert.

This chapter discusses advances to the above techniques and approaches using BRACE [21], a cloud-based, fully-automated tool for software defect prediction, reliability and availability modeling and analytics. BRACE includes a number of technical contributions to make software defect prediction more practical for every software development project. First, the tool unifies and automates the entire process of data extraction, pre-processing, core processing and post-processing. Second, the core data analytics engine of BRACE—SRGM—provides a robust, consistent, flexible, fast, statistically sound approach to defect prediction for any defect data set without human intervention. This is achieved by modeling the entire defect trend as a series of piece-wise exponential curves, and incorporating a mechanism to automatically detect when to transition from one curve to the next. Moreover, to enhance the accuracy, whenever available, the algorithm can also take as input feature arrival data (which is a measure of development effort). This allows the enhanced SRGM to provide defect prediction from the project planning phase through the internal testing phase and the customer site test and customer operation periods. Finally, BRACE exposes a user interface which displays the output generated from the core processing. It makes it easier for projects to understand the current progress towards quality improvement. To the best of our knowledge, this is the first attempt to propose a practical software defect prediction and reliability management solution that can be re-used across multiple teams at industrial scale.

1.3. Chapter overview

The rest of this chapter is organized as follows: In Section 2, we present the design, implementation and use cases of BRACE. We also describe datasets from two example projects that are used as references throughout the chapter. In Section 3, we detail an automated SRGM algorithm which is the core analytics engine of BRACE. The algorithm enhances traditional SRGMs by enabling accurate early defect prediction, which, as mentioned earlier, is a necessity for most projects. Sections 4 and 5 will discuss key post-delivery metrics: *customer defects* and *software availability* respectively. The Chapter is concluded in Section 5. Data sets from two large-scale software development projects from telecom products are used to illustrate the effectiveness of BRACE throughout the chapter.

2. BRACE system overview

2.1. System design

BRACE consists of three main processes: (1) *pre-processing*, (2) *core processing* and (3) *post-processing* as illustrated in **Figure 2**.

http://dx.doi.org/10.5772/intechopen.79839

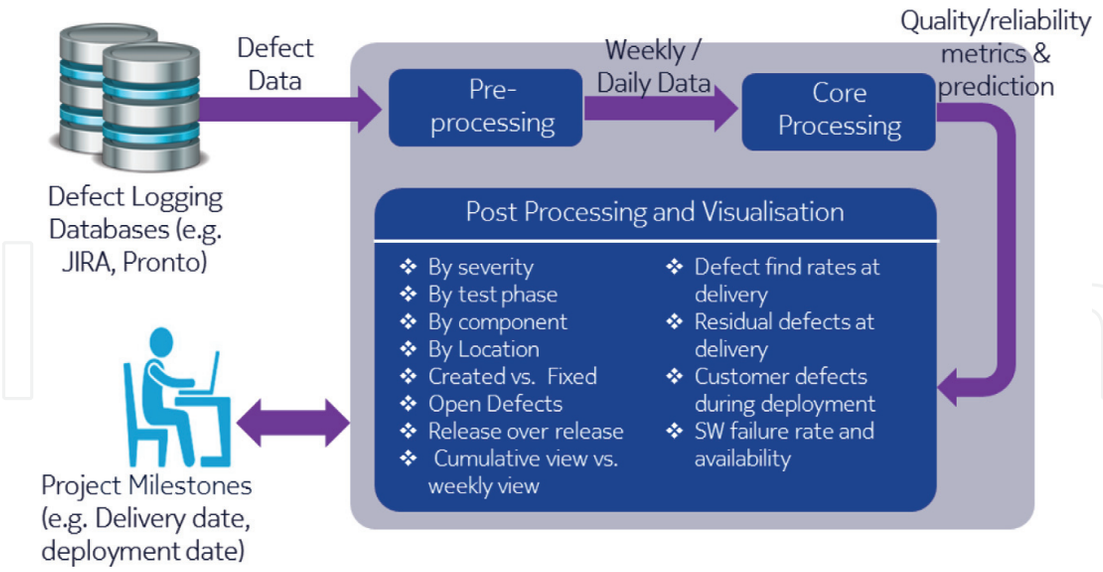

Figure 2. BRACE main processes.

1. The pre-processing step is made up of a generic data-collector which uses application programming interfaces (APIs) to collect data from various sources. As example, defect data can be obtained from Jira while other project-related information may be obtained from GitLab. The data is generally obtained as records of defects with the corresponding fields (such as creation date, resolution date, defect type, software release, etc.). Once the data is received for project, a number of processing steps are carried out. These might include—as necessary—*filtering* out specific defects based on any chosen field, *mapping* particular field/values, *grouping* defects based on any of the fields, etc. These processing steps may be generic for all projects, or may also be customized for a particular project. The output of this step is defect numbers aggregated over time (for example defects each day, 3-days, week[2]) for any desired *profile*. The *profiles* are project-specific analysis groupings such as defects related to a given software component, development location, severity, etc.

2. The core processing implements the optimization function which derives *maximum likelihood* estimates of the model parameters for a given set of defect data. It then—dynamically and automatically—identifies changes in the defect trend as new defect data is added. This is achieved by comparing the goodness-of-fit of the predicted values to actual data points. It eventually generates multiple curves to describe the entire defect trend using a piece-wise application of NHPP exponential models. The last curve is used for predicting total defects and residual defects at delivery. Detailed discussions on the SRGM used by the core processing step are presented in Section 3.

[2]While defect trend analysis is typically performed with weekly data, as we move towards continuous delivery, it is necessary to investigate the use of daily data. Therefore, the proposed tool has been applied to daily defect data from several projects, and has been able to produce results consistent with those from weekly data. Allowing the use of daily data is important as this allows SRGM to be applied in real time (daily), and hence help detect possible problems earlier.

3. Finally, the post-processing step takes as input the metrics from the core processing so as to generate various types of tables and charts, depending on project's needs. As an example, this step may be able to present outputs such as software failure rate, software availability & reliability, software annual downtime, defect rate, and predicted defects. Moreover, it also provides confidence limits for each of the calculated metrics. Therefore, the post-processing stage is aimed at providing answers (using a GUI) to the questions presented in **Table 1**. Answers to these use cases are the main motivations of the work done by software reliability experts, and by extension the main motivations of BRACE. As such, answers to the questions below will be provided throughout the rest of this chapter. We discuss details regarding these outputs and final processing steps in Sections 4 and 5, respectively.

2.2. Defect data sets

Defect data sets, called Projects A & B, are briefly described. Both projects represent large-scale software development from telecom products. **Figure 3** shows an example of a 3G & 4G wireless network system, where projects A and B can be found in the diagram.

2.2.1. Project A

This is a key wireless product, called RNC, which is responsible for the control and management of radio resources in a wireless network. It is a large-scale software development for a key 3G wireless telecommunication product with a high availability redundant hardware configuration. Code size varies from over 500 KNCSL (1000 non-commentary source lines) in earlier releases to less than 100 KNCSL as customers migrate from 3G to 4G systems. A traditional delivery scheme of one delivery per release was used. A major hardware platform change took place during the reported period, which resulted in redesigning the software architecture.

It takes advantages of hardware technological improvement, by introducing additional software redundancy as well as upgrading and re-designing the hardware platform with a pair of active-active processors. The new hardware platform supports multiple copies of a key

1.	How many defects are we going to find by delivery date?
2.	How many residual defects will remain at delivery?
3.	Are we ready for delivery?
4.	When do we expect the defect closure curve to catch up with the defect arrival curve?
5.	How many defects are we going to find after delivery?
6.	How does the defect curve of a new release compare to past releases?
7.	Which location/severity/component is generating the most defects?
8.	Can we combine inherent defects from previous releases, defects found within the current release and defects deferred to next release?
9.	Are we finding defects as expected?
10.	Can we adjust for DevOps continuous integration & delivery (CI/CD)?
11.	Can we predict software availability?

Table 1. BRACE use cases (motivating questions).

http://dx.doi.org/10.5772/intechopen.79839

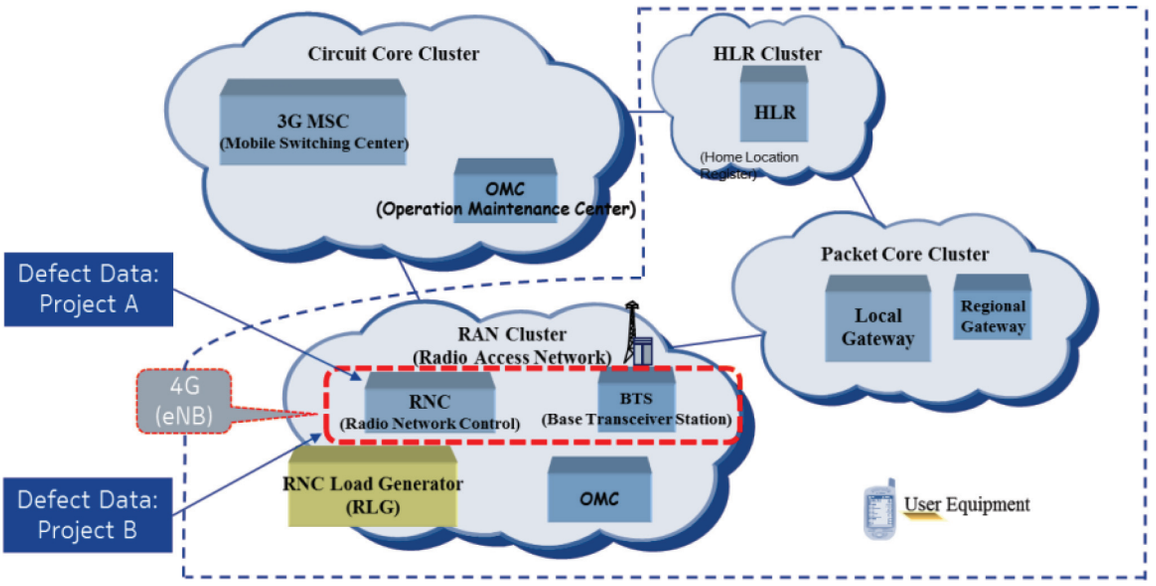

Figure 3. Sample 3G & 4G wireless telecom network system.

software component as many pairs of active-standby software configurations. As discussed in Section 4, this feature helped improve the capacity as well as the availability since the impact of failure software becomes very small due to a pair of active-standby. The quality impact of the major changes will be highlighted in Section 4. The data sets used in this chapter cover 11 releases over 5 years.

2.2.2. Project B

It represents a radio access part of the latest 4G mobile network technology. The two 3G key functions were combined into one, called eNB, to meet high data rate requirements. It performs the control and management of radio resources in a wireless network, plus radio frequency transmitters and receivers used to directly communicate with mobile devices. It is built on a highly complex hardware design and sophisticated software architecture. The software development is required to deliver many complex new features to meet demand in fast growing markets. Recent releases contain over 1 million non-commentary source lines (MNCSL) new features and deploy a new delivery scheme of multiple deliveries per release to satisfy the needs for additional features by multiple customers. Similar to Project A, this project also went through a major hardware platform change and drastic software redesign. The quality impact will be shown in Section 4.

3. Software reliability growth model (SRGM)

3.1. Automated SRGM

As earlier mentioned, exponential models assume that defects can be found and resolved at a constant rate [10]. While this results into a simple and flexible model with well understood

assumptions, it is not always the case that software development is stable, as sometimes changes have to be made to the processes. To ensure that defect prediction adapts to such trend changes, we apply the exponential prediction model to each wave of software defects. The idea of piecewise application of SRGMs is not exactly new. For example, a concept for evolving software content was originally discussed in [10]. This is the first time that we successfully formulated it mathematically, developed an innovative algorithm to automate the process and implemented it in a cloud environment.

The mathematical model is a non-homogeneous Poisson Process (NHPP) with a mean value function following an exponential model. The tool uses a piece-wise application of NHPP exponential models as illustrated in **Figure 4**. The NHPP assumption is used to implement the statistical method of maximum likelihood for estimating model parameters with the normal approximation confidence limits for a set of defect data. As new test defect data becomes available, we continuously monitor and predict residual (or remaining) defects at delivery. It then uses the last curve for predicting defects to be found after delivery to customer site.

To illustrate the model, consider a finite number, a, of defects such that each defect is found and removed by time, t, following a cumulative distribution function, $F(t)$. For a defect find process, $N(t)$, the probability of finding n defects by time t is in general expressed as a binomial distribution given in (1).

$$P\{N(t) = n\} = \binom{a}{n} F(t)^n [1 - F(t)]^{a-n} \tag{1}$$

In practice, the value of a is large, and therefore we can approximate (1) by a Poisson distribution with the mean value function, $m(t)$, as given in (2).

$$P\{N(t) = n\} = m(t)^n \exp\{-m(t)\}/n! \tag{2}$$

Note that $m(t) = aF(t)$ represents the average number of defects found by time t. An exponential model is described as an NHPP with the mean value function:

$$m(t) = a\{1 - \exp(-bt)\} \tag{3}$$

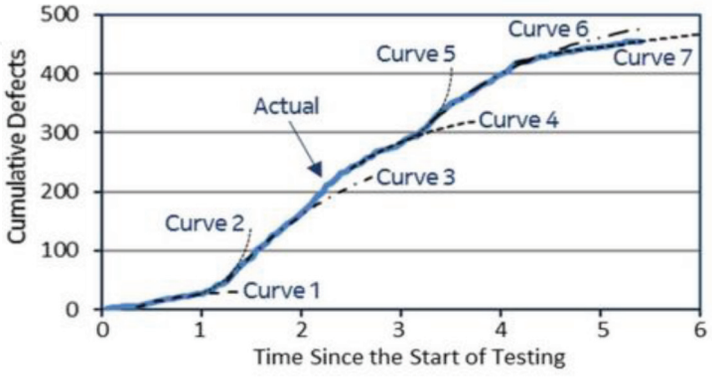

Figure 4. An example of a piece-wise application of NHPP model.

The parameters a and b represent total defects in the software and the rate at which each defect is found, respectively. Therefore, a total of a defects is assumed to be found according to an exponential distribution with a rate of b. Note that the parameter a represents the number of defects associated with each period for a case of piece-wise application. Since the mean value function is a function of time, it is called an exponential NHPP model. Taking the derivative of the mean value function we can derive the corresponding defect intensity function or defect rate, given in (4):

$$\lambda(t) = ab \exp(-bt) \tag{4}$$

It should be pointed out that if b is positive, m(t) converges exponentially, approaching to a positive value of a, and λ(t) decreases exponentially. This is a typical trend for reliability growth. As b approaches zero and a tends to infinity, $m(t)$ becomes a straight line and $\lambda(t)$ becomes constant, i.e., a stationary Poisson process. If both a and b are negative, both $m(t)$ and $\lambda(t)$ increase exponentially. Although most of the time b is positive, there are a few cases with b tending to zero during site test and in-service periods and b being negative in early test phases. Note that the basic assumption of a finite number of defects is violated if b is zero or negative. However, it will be useful in explaining different trends for individual test periods within the same release. We will discuss further with actual data later.

Answers to use cases (a), (b) & (c) in Section 2 can be illustrated using **Figure 5** as follows: SRGM predicts 3700 defects by the delivery date. Therefore, assuming that current date corresponds to week 19 (vertical blue line), we would expect to find 1200 more defects in the 11 weeks to delivery assuming the same test progress continues. Since SRGM predicts a total of 4500 defects and 3700 defects at delivery, the *residual defects* will be 800 (= 4500–3700). The percentage residual defect can be calculated as 18% (= 800 / 4500). Based on our experience, we have determined thresholds (the percentage of residual defects to total defects) and provide color codes that are indicative of software quality and the readiness to deliver. Specifically,

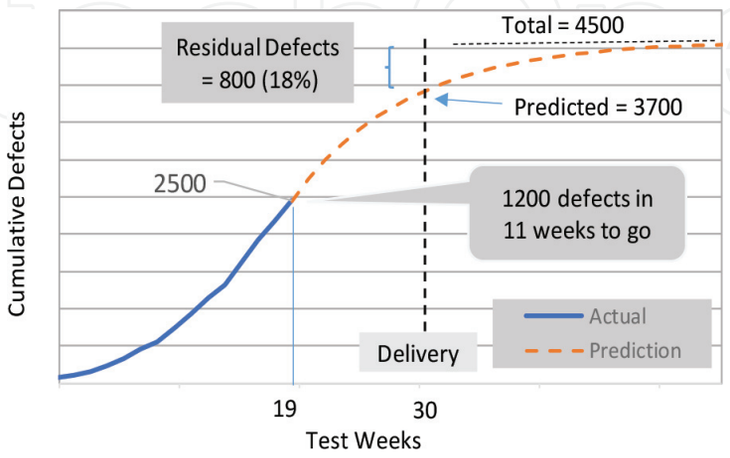

Figure 5. Example of software defect prediction.

readiness is given as green (implying 'good to go') if the threshold is less than 15%, yellow if it is between 15% and 25%, and red if it is greater than 25%. In this example, the software falls in the yellow range, indicating that some caution is needed if the project decides to proceed with the planned delivery.

As illustrated above, residual defects, which are derived from the defect arrival curve using SRGM, play a key role in software quality assessment in terms of delivery readiness. Our recommendation is that readiness is given as green (implying 'good to go') if the threshold is less than 15%, yellow if it is between 15 and 25%, and red if it is greater than 25%. In addition, it is important to track backlog defects at delivery, so as not to deliver known issues. Our recommendation is that all customer critical and major issues be resolved by delivery. We will address how to predict backlog defects in Section 3.2.2.

3.2. Enhanced SRGM: early defect prediction model (eDPM)

Typical SRGM techniques require defect data from the software test period. This limits their use during the early phases of software development during which it is usually necessary to make important (and time intensive) decisions (such as the level of staffing or amount of required testing or number of features to focus on) about the development process. Considering the industry trend towards very short software development lifecycles (i.e. agile development), it is essential to be able to make such decisions accurately very early on in the development phase. Specifically, in order to determine the staffing requirements for development and test activities during the early planning phase, many projects now need to understand what the defect find curve would look like during the internal test period. Therefore, early software defect prediction is needed for the early identification of software quality, cost overrun, and optimal development strategy.

We propose a novel method, eDPM, for predicting defect arrival curves based on the feature arrival curve during the planning phase. The feature arrival curve often gives the number of sub-features for each feature of the project, together with the times when each sub-feature is expected to be completed. Such information is usually available during the development planning phase of the software development life cycle. Specifically, eDPM involves using data from a previous release of the same product, together with the feature arrival curve for the upcoming release. In order to produce a reliability modeling approach that covers the whole development process, the eDPM approach has been integrated into BRACE as an enhanced SRGM.

3.2.1. Transformation functions

eDPM uses two transformation functions: one horizontal shift and one vertical shift. We have performed a statistical correlation study (e.g., quantile-quantile or Q-Q plot [22]) and found a very high correlation. **Figure 6** illustrates a Q-Q plot for Project B data. If the data points follow a straight line, it indicates a strong correlation between two factors being considered or statistically, the distributions of the two factors are the same. That is, both curves have similarity in shape.

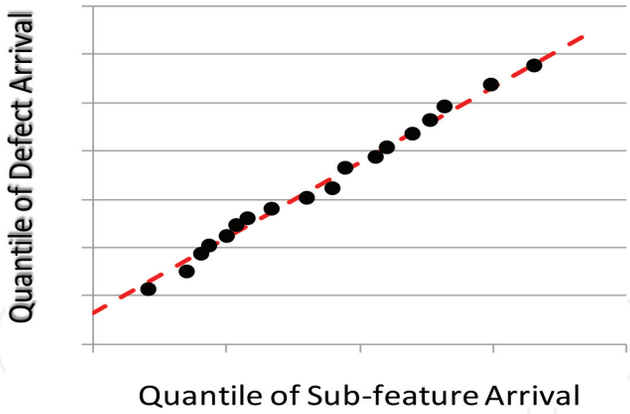

Figure 6. A sample Q-Q plot for project B release 5 data.

Let (x, y) represent a feature curve and (x_{new}, y_{new}) represent a defect arrival curve. We can move the feature curve to the right and closer to the defect arrival curve with the horizontal shift function in (5)

$$x_{new} = \alpha + \beta x \tag{5}$$

where α and β are parameters intrinsic to an individual release. The parameter α represents the average time to find α defects, and β represents the additional delay in the defect find process, likely due to a test resource constraint or critical bugs. Next, we use a simple form for the vertical shift as shown in (6).

$$y_{new} = \gamma\, y \tag{6}$$

The parameter γ is determined as a ratio of the feature count and the defect count used for the best fitted line in the Q-Q plot. It represents the number of defects per feature. Combining (5) and (6), we can transform the feature curve to represent the defect arrival curve. If previous release data is not available, we can use defect data from the initial test period. **Figure 7** demonstrates that feature ready curve is a good leading indicator for defect arrival curves. The feature arrival data is readily available for most projects.

3.2.2. Case studies

We will now provide four case studies to demonstrate the robustness of eDPM for practical uses. It should be pointed out that the team "feature" is used here in a generic sense to represent either sub-feature, epic, story, or sprint depending on the availability of metrics for individual projects. Similarly, the term "release" represents a set of features defined for each software delivery. The release content continues to evolve over the software lifecycle. It is important to continuously monitor the release content and adjust the transformation functions to improve the prediction accuracy. While out of scope for this chapter, we have recently developed an algorithm which automates the estimation of parameters as new feature and defect data becomes available.

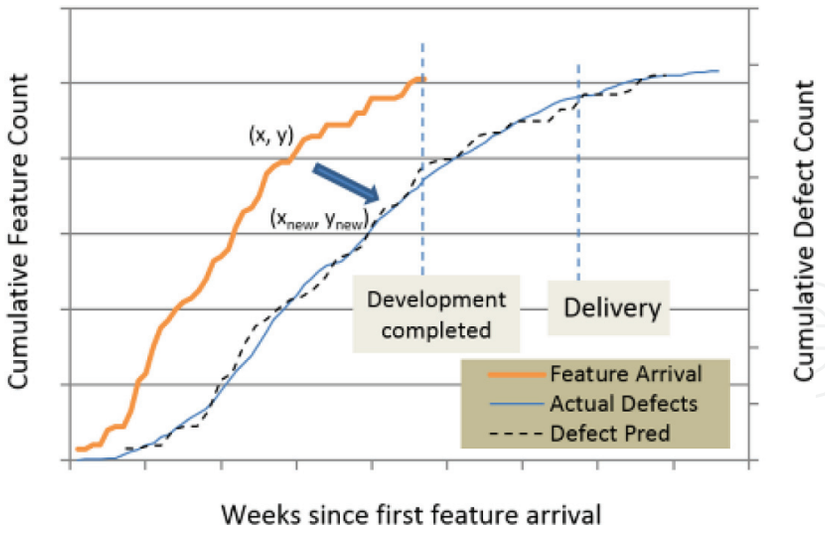

Figure 7. eDPM defect arrival curve prediction based on feature arrival curve.

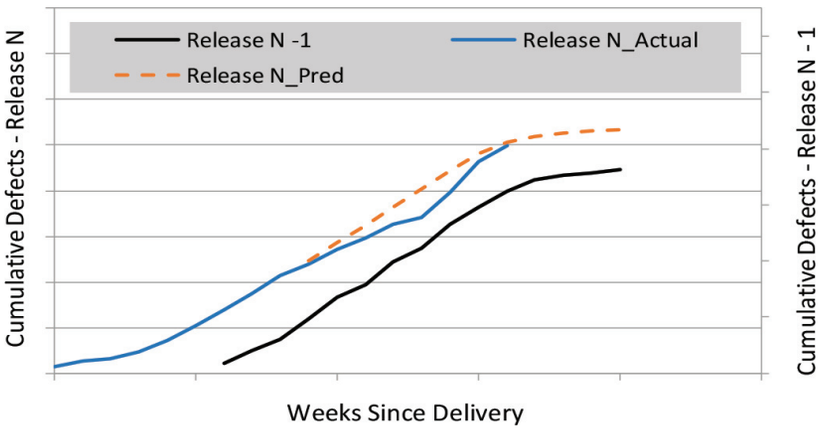

Figure 8. eDPM case study #1: Previous release data.

Case Study #1—Previous release data: This case study considers a project without feature ready data available. We use previous release data, called Release N – 1, to predict the defect arrival curve for Release N. Using the transformation functions described in Section 3.2.1 and the test defect data from Release N, we can predict the defect arrival curve, as shown in **Figure 8**. Actual data is overlaid and compared against the predicted arrival curve. The several weeks (between −10 and −5) the actual data was not following the predicted curve were due to a few critical issues slowing down the test progress. Once they were cleared with fixes, the test progress was quickly recovered and the defects started to come in as expected.

Case Study #2—Test cases executed: This case considers a project without accurate data on feature ready dates, but having a good record of test cases executed prior to handing the features over to the test team. For our eDPM purpose the test case data is considered equivalent to the feature ready data. **Figure 9** demonstrates near-perfect prediction of the defect arrival curve using test cases executed.

http://dx.doi.org/10.5772/intechopen.79839

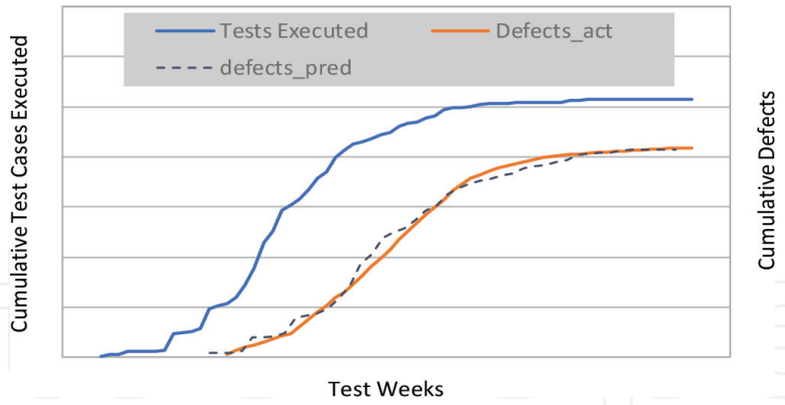

Figure 9. eDMP case study #2: Test cases executed vs. test defects.

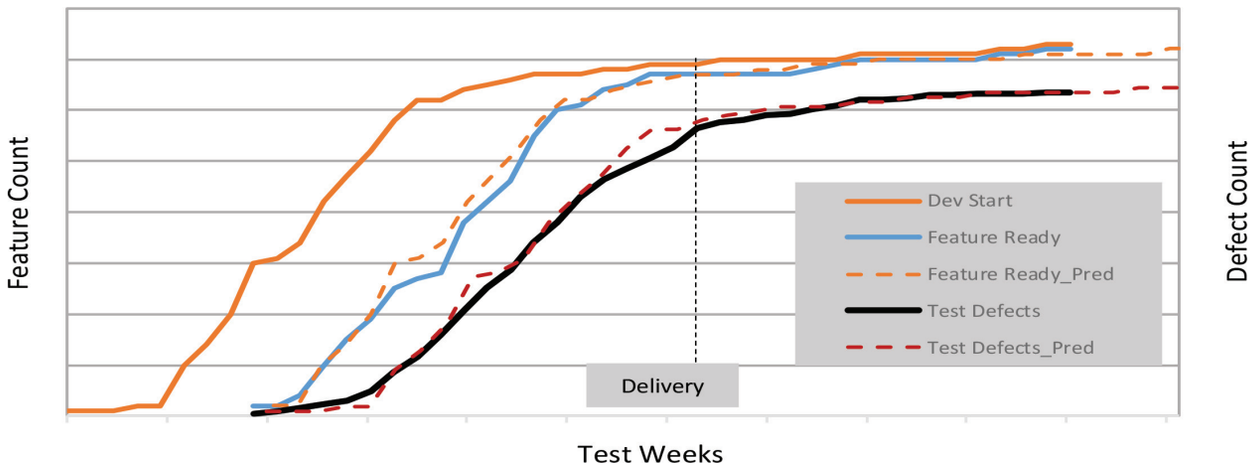

Figure 10. eDPM case study #3: Development start vs. feature ready vs. test defects.

Case Study #3—Feature development start data: This considers a project with a good record of feature development status. We first use feature ready dates to predict the defect arrival curve. As expected, it gives a good prediction. The next task is to evaluate if we can use the development start dates to predict the feature ready curve and the defect arrival curve. This is important to help a project to plan the development and test resources for both feature ready dates and test defects. In this case we apply the transformation functions in two phases, i.e., one for predicting feature ready curve and the other for predicting the defect arrival curve. As illustrated in **Figure 10**, the predictions are remarkably accurate for both cases.

Case Study #4—DevOps CI/CD story points completed: The case considers a DevOps continuous integration & delivery project with a delivery interval of 2 or 3 weeks. The project implements a full Agile development process. **Figure 11** shows cumulative views of story points and integration test defects in two different vertical scales. The vertical lines represent individual release dates. A user story is a very low-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it. A story point is a measure of the effort required to implement a story. It is a relative measure of complexity, albeit a subjective one. However, if it is done in a

consistent manner, it can be used as a good leading indicator for predicting defects, as shown in **Figure 11**.

It was later confirmed that there were two major process changes made during the reported period. As eDPM was applied to the data (Transformation #1), we were able to identify the trend change after several months where the transformation is no longer valid. It turns out that the trend change occurred when a major process change was made. Another set of transformation functions, called Transformation #2, were then used. The predicted values are very closely matching with actual defect data. Several months later, we encountered another trend change, which turns out to be caused by another major process change. We then used another transformation #3. With the successive use of eDPM we demonstrated that defects can be predicted with reasonable accuracy for the entire reported period.

Another benefit of eDPM is to help quantify the process improvement. One of the parameters, γ, as described in Eq. (6), represents defects per story point in this case study. By comparing the values of γ between two transformation periods, we can calculate the relative change in γ values. This relative change represents the percent of improvement due to the process change. To further provide the significance of this benefit, we were able to quantify an improvement of 10 and 70% for the first process change and the second, respectively.

Case Study #5—Defect closure data: This case study we consider a project with both defect arrival and closure data in addition to sub-feature arrival data. This project is still in early test phase but project management wants to know whether the defect backlog can reach zero at the delivery date. First, we predict the defect arrival curve based on the sub-feature arrival data and actual defect arrival data so far using eDPM. We then predict the closure curve based on the predicted arrival curve using eDPM. By combining the predicted arrival and closure curves we can now calculate the defect backlog by subtracting the closure curve from the arrival curve. **Figure 12** shows the predicted arrival and closure curves, along with the predicted defect backlog curve. The project can now see some actions to be taken to improve the backlog curve to get closer to zero at the delivery date.

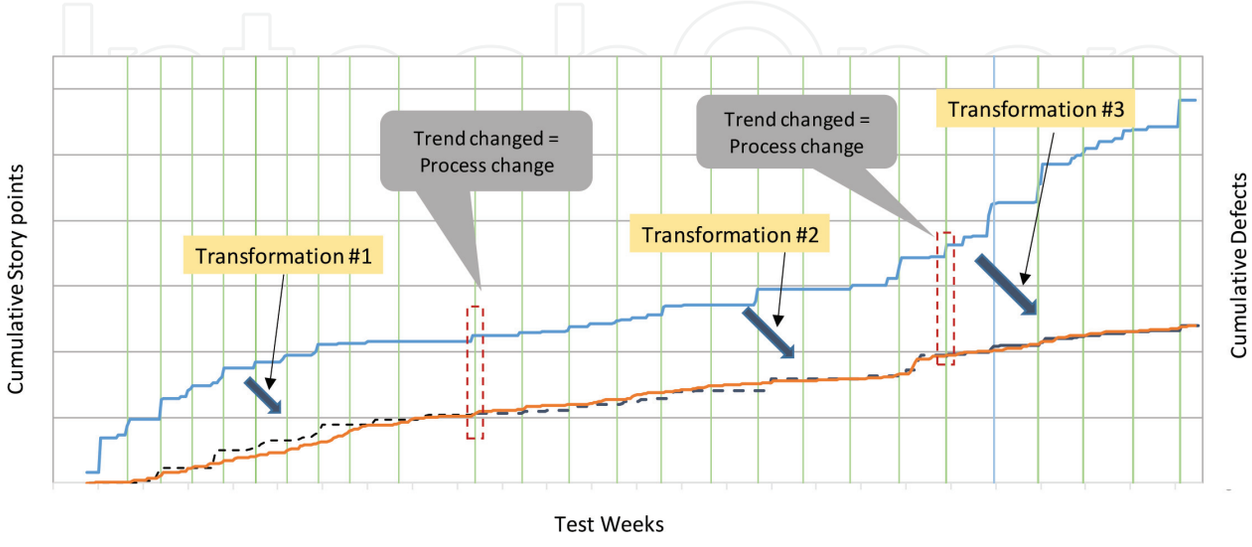

Figure 11. eDPM case study #4: Story points vs. integration test defects.

http://dx.doi.org/10.5772/intechopen.79839

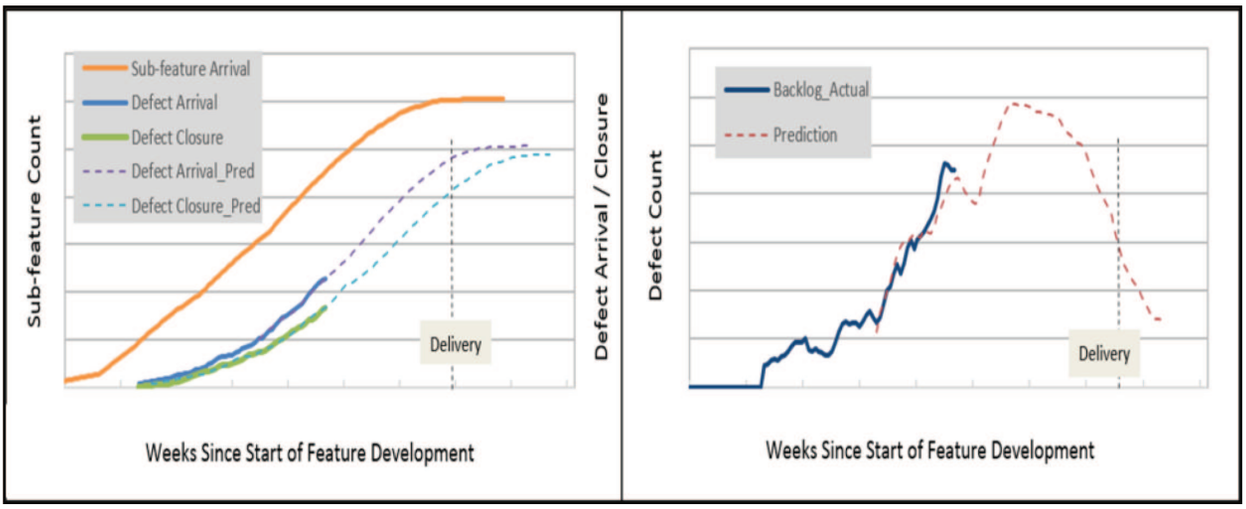

Figure 12. eDPM case study #5: sub-feature, defect arrival/closure/backlog.

4. Customer defect prediction

In Section 3, we demonstrated that the last curve prior to software delivery represents the final product from which the total number of defects and residual defects can be calculated. Previous release data or historical data from other projects will be helpful for determining the percent of delivered defects to be found during the operation period. See [16] for detailed discussions.

The assumption that the defect curve can be extended from the development phase into the operational phase (e.g. [23–25]) does not hold in practice, as there are usually discontinuities due to changes in the intensity of testing, as well as operational conditions not always being exactly the same as test environments [7].

To highlight the procedure and results we will use defect data taken from Project B. **Figure 13** shows a cumulative view of customer defect prediction. Note that the curve should be always above the actual data after delivery. The difference between the curve and the last actual data

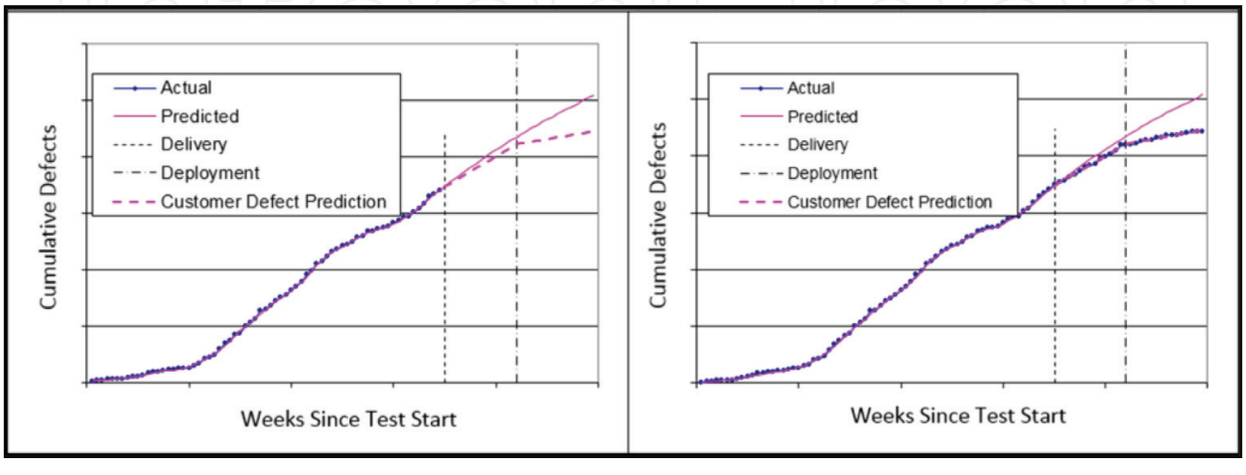

Figure 13. Cumulative view of project B customer defect prediction vs. actual.

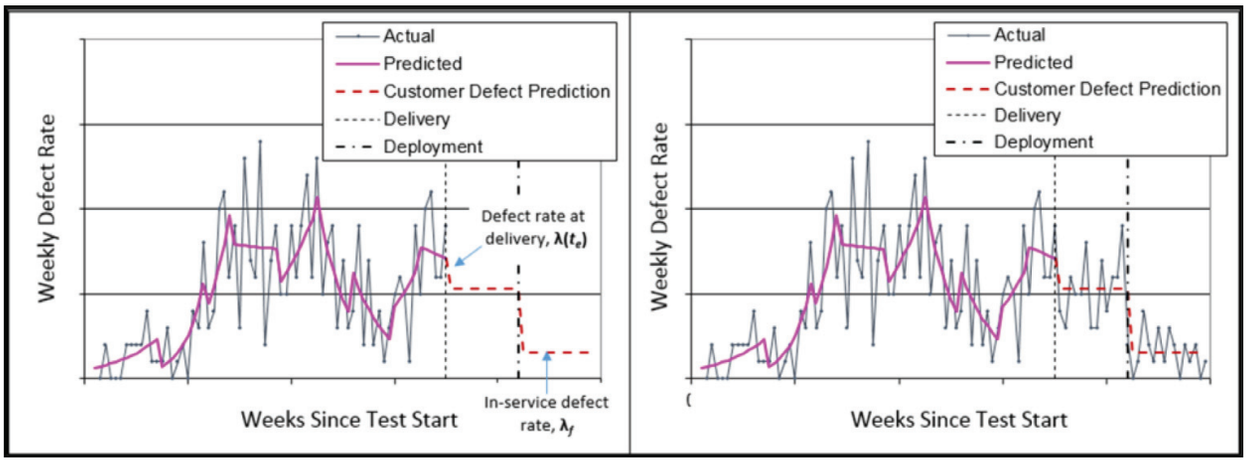

Figure 14. Weekly view of project B customer defect prediction vs. actual.

point indicates the defects not found in this release and they will become a part of the next release. That is, not all delivered defects will be found during the operation period. It also demonstrates that actual data follows as predicted, indicating the importance of historical data in predicting post-delivery defects.

Figure 14 illustrates the difference in defect rate, $\lambda(t_e)$, at the end of test phase, t_e, and during the operation period, λ_f. It can be observed that there is a difference in defect rate, likely due to differences in the intensity of testing during the two periods, as well as possible differences between a test environment and a field operational state.

5. Software failure rate, availability, and reliability

In recent years many product suppliers have been implementing complex software-controlled systems with a large number of software features on a short development schedule. In the telecom industry, a critical customer operational issue is on system performance, especially in terms of system outages impacting the service availability for their end users. As a result, service providers frequently ask their product suppliers for software reliability and availability measurements. In this section, we discuss the relationship between software failure rate, availability, and reliability.

5.1. Software failure rate

Field outage measurements are required for telecom products by TL9000 [26], which is a quality management system (QMS). It standardizes the quality system requirements for the design, development, delivery, installation, and maintenance of telecom products and services. It defines the reliability in terms of SO3 (service outage frequency) and SO4 (service outage duration) metrics. As demonstrated in [16] the defect find process during the operation period maybe modeled as a stationary Poisson process. It also follows that the rate of software failure (or outage) rate for each release can be modeled as stationary Poisson process. Consider a software release with a failure rate λ and defect rate λ_f. It is worth noting that λ is usually

http://dx.doi.org/10.5772/intechopen.79839

measured in terms of failures per year. The defect conversion factor may be expressed as shown in Eq. (7)

$$d = \frac{\lambda}{\lambda_f} \tag{7}$$

Reliability and availability and among the key factors that are used to define the quality of software in practice. In what follows, we formulate mathematical representations for both these factors.

5.2. Availability

The availability of software can be expressed using cycles of uninterrupted working intervals (Uptime), followed by a repair period after a failure has occurred (Downtime) using (8).

$$A = \frac{Uptime}{Uptime + Downtime} = 1 - \frac{Downtime}{Uptime + Downtime} \tag{8}$$

Considering that availability is typically evaluated over a 1 year period, $Uptime + Downtime = 1\ year = (60 \times 24 \times 365)$ minutes. Therefore, as an example, to achieve system availability of 5 9's (i.e. A = 99.999%) the maximum allowed downtime would be 5.26 minutes/year.

5.3. Reliability

On the other hand, software *reliability* is the probability that the software has not failed after a time period t. Therefore, reliability is a function of t, and can be denoted as R(t). R(t) is typically modeled using an exponential distribution in which the parameter is failure rate λ as shown in Eq. (9)

$$R(t) = 1 - \exp(-\lambda t) \tag{9}$$

It is important to note that while both reliability and availability are a measure of software quality, they have different technical meanings. In particular, availability is determined by both uptime and downtime, while reliability is only influenced by uptime. This implies that two software releases or systems having the same failure rate, would have the same reliability, but might have different availabilities. Achieving a high availability generally requires having automated ways of recovering from failures, for example, through redundancy or rebooting, so that the downtime is minimized. Software failures for which the system is able to automatically recover are known as covered failures. On the other hand, if a system fails to automatically detect and/or recover from a failure, such a failure is known as an uncovered failure, and usually leads to customer perceived defects. In systems where recovery time is significant, a coverage factor – the proportion of all failures that are covered failures – is defined. However, in most practical applications, it requires specialized tools to determine covered failures. Therefore, typical failure counts usually only consider the uncovered defects.

5.4. Discussion

In what follows, we use (anonymised/scaled) data from project A to demonstrate the various aspects of software failure, reliability and availability, together with the predictions that are

carried for the same. The data compares multiple releases of a software product over multiple years. Outage data represents unplanned, customer-reported, and uncovered failures, including full and partial outages. The outages were collected across a deployment of over 400 systems. The monthly outage count is annualized and normalized by the number of deployed systems as outages/year/system, which is equivalent to the failure rate. In the same way, the monthly outage downtime is annualized and normalized by deployed systems as downtime/year/deployed system. It should be noted that the downtime duration of each outage is discounted by percentage impact (i.e. 100% being a full outage), using the TL 9000 counting rule.

In **Figure 15**, we show the predicted software reliability as a function of failure rate (on the left) and software availability as a function of annual downtime (on the right). The following observations can be made:

- From one release to another, the actual data generally lies within the 90% confidence limits for both availability and reliability. This is testament to the accuracy of the generated predictions.
- Over time, from one release to the next, we can observe a continuous improvement in software availability and reliability. This is not surprising since it takes time and increased effort to enhance software design, development and test practices.
- There is a slight deterioration in reliability and availability at R5, corresponding to a change in hardware, but these quickly improve again after that. This can be explained by the need to re-design the software, but also demonstrates the important effect hardware can have on the quality of software, i.e. sometimes significant long term improvements in software quality may only be achieved through changes in hardware.
- It is worth observing that predicting availability is generally more difficult than predicting reliability. This is due to the fact that availability is affected by the downtime while reliability is not. In addition, as software development teams get used to a product from one release to the next, they get used to the system, and therefore, are normally able to significantly reduce the average system downtime.

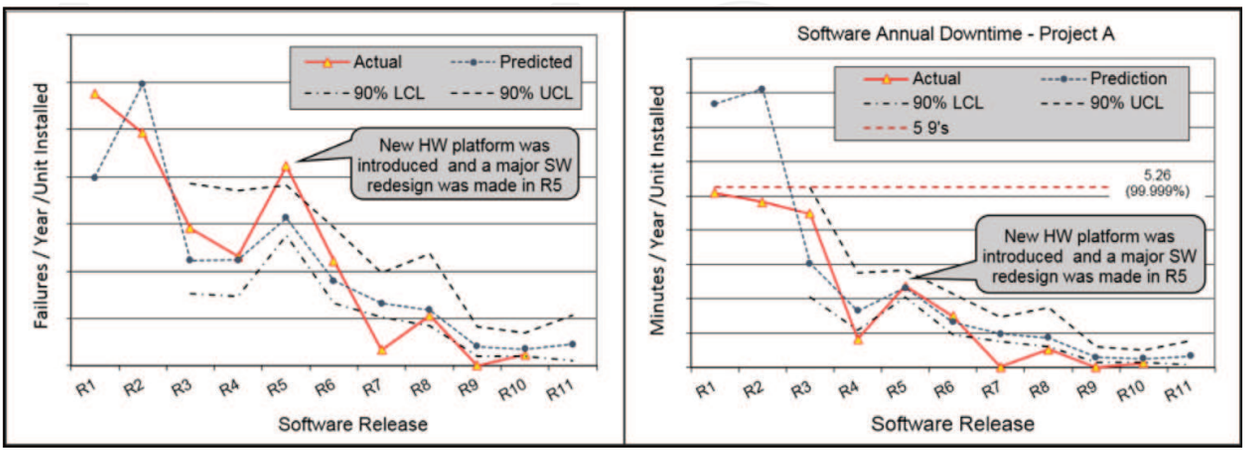

Figure 15. Release-over release software reliability and availability prediction—Project A.

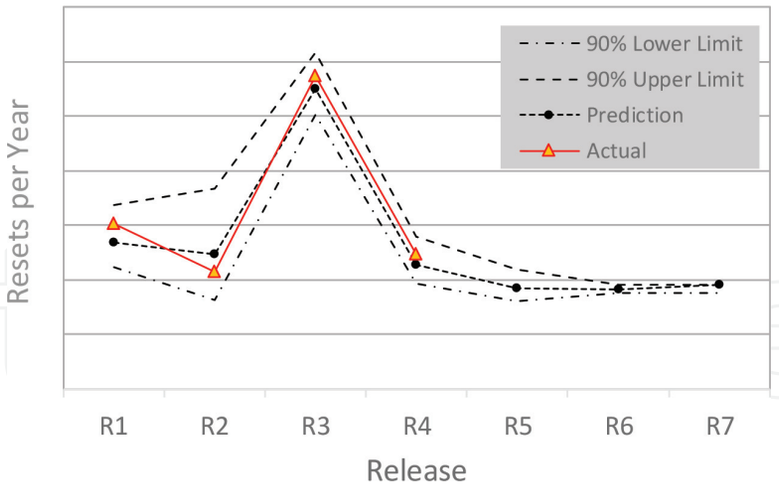

Figure 16. Software failure rate prediction—Project B.

Finally, we applied the method to Project B. In this project, it is not practical to collect the system downtime in the field due to the nature of the product. However, customers are concerned about resets. Therefore, the focus is on the number of unplanned autonomous resets. **Figure 16** summarizes the annual reset rate with prediction and actual data over several releases. The predicted values are remarkably close to actual data and within the 90% limits. Although actual downtime is not available, we can use reset time measured in the lab to calculate the reset-based availability using the reset rate prediction.

6. Implementation

Figure 17 shows the implementation of BRACE. The tool is made up of multiple application programming interfaces (APIs), each of them connecting to a defect logging database (such as JIRA). Defect data is collected from the defect databases in real-time and pre-processed by a computer program (in Python) before being stored into a cloud-based, shared database used by the system. The SRGM algorithm (which is written in Python) then performs the core processing, providing a consistent, fast, flexible, robust, and statistically sound result. Using the output of from core processing, we have also created a unified graphical user interface (GUI) onto which a wealth of software quality metrics are presented to users. While in the current implementation all components of the tool are hosted in a virtual machine running in openstack, it is possible to have them also running in a dedicated server if needed.

As an example use case, for a given project, a number of input parameters are required for the tool. Such inputs include the project milestones, the require changes in defect rate before and after deployment, and a number of assumptions based on expert knowledge of both the product and development process.

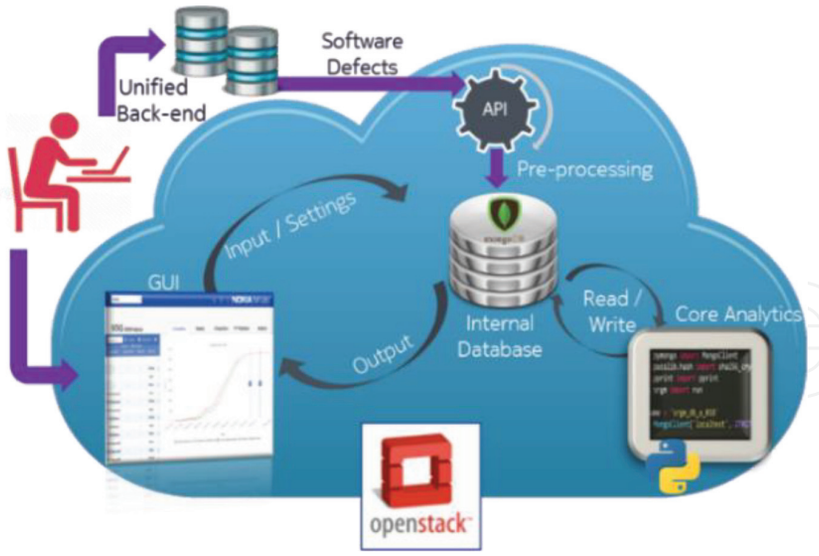

Figure 17. Design and implementation of a cloud-based BRACE.

7. Conclusion

In this chapter we presented a practical approach to software defect prediction, which helps assure the delivery of high quality software. An innovative cloud-based analytics tool, BRACE, was introduced which automates the entire process of data extraction, pre-processing, core processing, and post-processing, combined with a user interface. It no longer relies on the use of a spreadsheet and generates prediction in real-time, which can be shared with any members of a project. SRGM is the core analytics engine which implements technical breakthroughs in this area. It provides a robust, consistent, flexible, fast, statistically sound approach to defect prediction for any defect data sets without human intervention. The enhanced version of SRGM incorporates feature arrival data to provide defect prediction throughout the lifecycle of each release with much improved accuracy. We also demonstrated the method for predicting customer defects and software availability during the operation phase, which should be the basis for software quality assurance. We demonstrated the effectiveness of the approach using data sets taken from telecom development projects, varying from traditional development to DevOps CI/CD with full agile development. This approach can be easily applied to any software development projects.

Author details

Kazu Okumoto[1]*, Rashid Mijumbi[1] and Abhaya Asthana[2]

*Address all correspondence to: kazu.okumoto@nokia-bell-labs.com

1 Nokia Bell Labs, Dublin, Ireland

2 Nokia Bell Labs, Westford, USA

References

[1] Mijumbi R, Serrat J, Gorricho JL, Bouten N, De Turck F, Boutaba R. Network function virtualization: State-of-the-art and research challenges. IEEE Communications Surveys & Tutorials. 2016;**18**(1):236-262

[2] Ozakinci R, Tarhan A. The role of process in early software defect prediction: Methods. In: Attributes and Metrics. Cham: Springer International Publishing; 2016. pp. 287-300. DOI: 10.1007/978-3-319-38980-6 21

[3] Martin LS. Probabilistic models for software reliability prediction. In: Freiberger W, editor. Statistical Computer Performance Evaluation. Academic Press; 1972. pp. 485-502, ISBN 9780122669507, https://doi.org/10.1016/B978-0-12-266950-7.50029-3. (http://www.sciencedirect.com/science/article/pii/B9780122669507500293)

[4] Jelinski Z, Moranda PB. Software reliability research. In: Feiberger W, editor. Statistical Computer Performance Evaluation. New York: Academic; 1972. pp. 465-484

[5] Schick GJ, Wolverton RW. Assessment of software reliability. In: Proceedings of Operations Research. Wurzburg-Wien. 1973. pp.395-422

[6] Schneidewind NF. Analysis of error processes in computer software. In: Proceedings of the International Conference on Reliable Software. IEEE Computer Society; 1975. pp. 337-346

[7] Musa JD. A theory of software engineering and its application. IEEE Transactions on Software Engineering, SE-1. 1975;**3**:312-327

[8] Goel AL, Okumoto K. Time-dependent error-detection rate model for software reliability and other performance measures. IEEE Transactions on Reliability. 1979:206-211

[9] Yamada S, Ohba M, Osaki S. S-shaped reliability growth modeling for software error detection. IEEE Transactions on Reliability. 1983:475-478

[10] Musa JD, Iannino A, Okumoto K. Software Reliability: Measurement, Prediction, Application. New York: McGraw-Hill; 1987

[11] Musa JD. Operational profiles in software-reliability engineering. IEEE Software. 1993: 14-32

[12] Wallace D, Coleman C. Application and improvement of software reliability models. In: Hardware and Software Reliability (323–08). Software Assurance Technology Center (SATC); 2001

[13] Okamura H, Dohi T, Osaki S. A reliability assessment method for software products in operational phase—Proposal of an accelerated life testing model. Electronics and Communications in Japan. 2001:25-33

[14] Jeske DR, Zhang X, Pham H. Adjusting software failure rates that are estimated from test data. IEEE Transactions on Reliability. 2005:107-114

[15] Zhang X, Pham H. Software field failure rate prediction before software deployment. Journal of Systems and Software. 2006:291-300

[16] Okumoto K. Experience report: Practical software availability prediction in telecommunication industry. In: Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering (ISSRE). 2016. pp. 331-342

[17] Asthana A, Okumoto K. Integrative software design for reliability: Beyond models and defect prediction. Bell Labs Technical Journal. 2012;**17**(3):39-62

[18] Lyu MR. Handbook of Software Reliability Engineering. New York: Computer Society Press, Los Alamitos and McGraw-Hill; 1995

[19] Okamura H, Dohi T. SRATS: Software reliability assessment tool on spreadsheet. In: Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE 2013). IEEE CPS; 2013. pp. 100-117

[20] Wikipedia. List of Software Reliability Models. https://en.wikipedia.org/wiki/List_of_software_reliability_models

[21] Okumoto K, Mijumbi R, Asthana A. Brace: Cloud-based software reliability assurance. In: 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), vol. 00. Oct. 2017. pp. 57-60. [Online]. DOI: ieeecomputersociety.org/10.1109/ISSREW.2017.48

[22] https://en.wikipedia.org/wiki/Q%E2%80%93Q_plot

[23] Kimura M, Toyota T, Yamada S. Economic analysis of software release problems with warranty cost and relaibility requirement. Reliability Engineering & System Safety. 1999: 49-55

[24] Yang B, Xie M. A study of operational and testing reliability in software reliability analysis. Reliability Engineering & System Safety. 2000:323-329

[25] Ukimoto S, Dohi T. A software cost model with reliability constraint under two operational scenarios. International Journal of Software Engineering and Its Applications. 2013; **7**(1)

[26] TL 9000 Measurements Handbook, Release 4.0, 2007, Quality Excellence for Suppliers of Telecommunications (QuEST) Forum, http://www.tl9000.org/handbooks/documents/ProdCat_Tables_4-0.pdf