Assessing the Maturity of SDN Controllers with Software Reliability Growth Models

Petra Vizarreta, Kishor Trivedi, Bjarne Helvik, Poul Heegaard, Andreas Blenk, Wolfgang Kellerer, and Carmen Mas Machuca

***Abstract*—In Software Defined Networking (SDN) critical control plane functions are offloaded to a software entity known as the SDN controller. Today's SDN controllers are complex software systems, owing to heterogeneity of networks and forwarding devices they support, and are inherently prone to bugs. Our previous work showed that Software Reliability Growth Models (SRGM) can model the stochastic nature of bug manifestation process of the ONOS open source controller. In this article we focus on different applications of our SRGM framework crucial for an efficient management of SDN-based networks. We provide guidelines for network operators to decide when the controller software is mature enough to be deployed in operational environment, based on the reliability requirements of network applications, and quantify the marginal benefits of the prolonged testing phase on the software quality. We show how the accuracy of software reliability prediction in the early phase of the software lifecycle can be improved by extrapolating the behaviour of previous controller software releases, demonstrated in the example of twelve ONOS releases. We also propose software maturity metrics, that can be used by operators to discriminate between the competing SDN controller designs, i.e. ONOS and OpenDaylight, when software reliability is a major concern.**

***Index Terms*—Software Defined Networking, software maturity, software reliability prediction, Software Reliability Growth Models.**

I. Introduction

A. Motivation and problem definition

Software Defined Networking (SDN) is an architectural concept of decoupling control and data plane, by outsourcing all control plane decisions of forwarding devices to a logically centralized software entity known as the SDN controller. The SDN controller assumes the role of the network operating system, providing an integrated interface towards the forwarding devices, switches and routers, which significantly simplifies the network management and augments its programmability [1]. The controller monitors the state of the network by gathering the statistics from forwarding devices, makes the global routing decisions, and reacts on the events, such as link congestion or switch failure. In order to fulfil the long list of tasks, today's production-grade controllers, ONOS [2] and OpenDaylight [3], have grown to be rather complex pieces of software, consisting of more than a million lines of code. Such a large and complex[1] software inevitably contains bugs, that may disrupt the network operation and corrupt its performance, when triggered. Recent study on the hazards in SDN-based Google network infrastructure [4] reported that software bugs contributed to more than 33% of the high impact failures documented in post-mortem reports, which they attribute mainly to a high velocity of network evolution and the need to keep up with the growing user traffic and demand for new features and services. Another large-scale study by Microsoft [5] on root causes of customer-impacting incidents in their production networks reports similar results, and shows that software bugs contributed to 36% of critical outages, being major problem, way ahead of hardware failures and human errors.

Despite the magnitude and ubiquity of software failures, there is a lack of the tools to quantify software maturity, and predict the risk of the software related outages in SDN. The performance reports and benchmarks on SDN controllers are still limited to scalability and latency related metrics, such as flow burst install throughput or flow reroute latency. The reliability of the controller software, which is still a big concern and a major obstacle for the wide spread adoption of SDN in commercial telecom and industrial networks [6], is addressed only by a limited number of studies [7]–[9].

B. Theoretical background and problem solution

Software reliability growth modelling is a statistical framework, used to estimate the reliability of the software components in their operational phase, based on the bug manifestation reports from the testing phase. During the testing and early operational phase of the software lifecycle the faults are detected and removed, which eventually leads to reliability growth. The core idea behind SRGM is to describe the fault detection and fault resolution as stochastic processes, whose parameters can be estimated from the empirical data, i.e. the history of the previous bug manifestations. Once the best stochastic model to describe the data is found and parametrized, it can be used to estimate reliability metrics, such as *residual bug content*, *failure intensity* or *expected time until the next failure*, and conditional *software reliability*.

P. Vizarreta, A. Blenk, W. Kellerer, C. Mas Machuca are with the Chair of Communication Networks, Technical University of Munich, Germany, e-mail: (petra.vizarreta@lkn.ei.tum.de.

B. Helvik, P. Heegaard, are with the Department of Information Security and Communication Technology, Norwegian University in Science and Technology, Norway.

K. Trivedi is with Department of Electrical and Computer Engineering, Duke University,USA.

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671648 (VirtuWind), COST Action CA15127 Resilient communication services protecting end-user applications from disaster-based failures (RECODIS) and was supported in part by US NSF Grant number CNS-1523994, CELTIC EUREKA project SENDATE-PLANETS (Project ID C2015/3-1) and the German BMBF (Project ID 16KIS0473).

[1] As a reference, the latest Linux kernel has around 20 million lines of code.

Moreover, SRGM enable the operators to estimate how reliability metrics change over time, as the software matures. In essence, the software reliability metrics captures the relationship between the testing effort and the software quality, which is highly relevant for the developers and operators customizing the existing open source solutions. With SRGM the risk of the software outages in a given period of time can be predicted with the high accuracy, providing useful guideline for the operators of SDN networks to take the calculated risk and chose the best software adoption time, based on the reliability requirements of their network applications. The pertinence of SRGMs was already recognized by Network Function Virtualization (NFV) community, which has already included in the guidelines for the assessment end-to-end reliability [10].

C. Our contribution

Our study aims to provide a framework to assess the maturity of SDN controllers, from the perspective of software developers and network operators. We extend our previous work [9], which focused on applicability of SRGM and software reliability of ONOS open source controller. In this article we explore different applications of our framework crucial for an efficient management of SDN-networks. The workflow steps: data collection, model selection, evaluation of reliability and management KPIs, are illustrated in Fig. 1, highlighting our main contributions:

1) Data collection: We gather the empirical data, i.e. cumulative number of detected and resolved bugs, from public issue trackers and provide a high level statistics that can be deduced from such data, e.g. distributions of time between failures and time to resolve a bug.
2) Model selection: We find the best SRGM to describe the stochastic behaviour of bug detection and bug resolution processs in SDN controllers. We show that the bug detection process can be described with the class of S-shaped SRGMs, and further propose new class of models for fault correction process, as well as their corresponding fitting technique.
3) Evaluation of reliability and management KPIs: We show how the software reliability metrics can be used to assess the quality of the controller software over time, and provide the guidelines for the optimal software release and adoption time. We further propose two novel applications relevant for the SDN community: i) early prediction of software reliability based on the previous releases and ii) software maturity metrics as a comparison criteria between alternative software solutions.

The rest of the article is organized as follows. Section II provides an overview of the related work on software reliability growth modelling. A theoretical background of SRGM framework is presented in Section III. In Section IV the gathering, processing and analysis of the bug reports is discussed. The model selection is discussed in Section V, while Section VI presents the applications of software reliability prediction for network management community. We conclude the paper with a summary and an outlook for the future work.

II. Related work

Software reliability growth modelling has been widely used to estimate and predict the reliability of the software, and in the past, many different models have been proposed. A good overview of different classes of reliability growth models, together with their inherent assumptions and input data requirements, can be found in [11]. In this section we present the most relevant models, methods and tools for the fitting of model parameters, as well as the applications of the software reliability assessment.

Bug detection: The applicability of SRGMs for the modeling, analysis and evaluation of software reliability of open source products was demonstrated in several case studies. Zhou et al. [12] showed that the Weibull distribution can describe well the bug manifestation rate for eight unnamed software projects. Rahmani et al. [13] confirmed this result by analyzing the bug reports for several popular big open source projects, such as Apache HTTP server, Eclipse IDE and Mozilla Firefox. Rossi et al. [14] studied failure occurrence pattern across different releases of Mozilla Firefox, OpenSuse and OpenOffice.org. All studied releases showed the learning curve pattern, where the fault detection rate is slow at the beginning until the community gets familiar with the product, then it increases rapidly until only very few faults, whose discovery is difficult, remain in the code. This effect is captured well with S-shaped models. Syed et al. [15] and Ullah et al. [16] studied the difference between the closed and open source software with the inconclusive results. In this work we compare eight most widely used SRGMs for the fault detection process [17]–[24] in terms of their ability to describe the empirical data.

Bug removal: The majority of the SRGM models assume that once the bug is detected, it is corrected immediately, that the debugging as always successful and without introduction of new faults. A number of studies have modelled different aspects of imperfect debugging [25]–[30]. Wu et al. [25] described the fault resolution as a delayed fault detection process, Pham et al. modelled the introduction of the new faults [26], while Huang et al. [27] also include the changes in debugging effort. Kapur et al. [28] generalized this result and proposed unified approach to model the fault resolution process, when both fault detection and fault removal are Non-Homogeneous Poison Processes. Gokhale et al. [29] applied the Non-Homogeneous Continuous Time Markov Chains (NH-CTMC) to model the impact of arbitrary debugging policy, while the study by Okamura and Dohi [30] modelled the time dependency between the fault detection and fault correction processes as a correlation. Comprehensive models have a large number of parameters that have to be estimated, while the number of data samples in the historical reports is often very limited (as in the case with ONOS SDN controller), which increases the risk of overfitting the data, as well as the sensitivity of parameter fitting to the noise in the data. In order to balance between the model accuracy and generalizability, we propose a simpler class of models, based on the framework presented in [28], together with their corresponding fitting procedure.

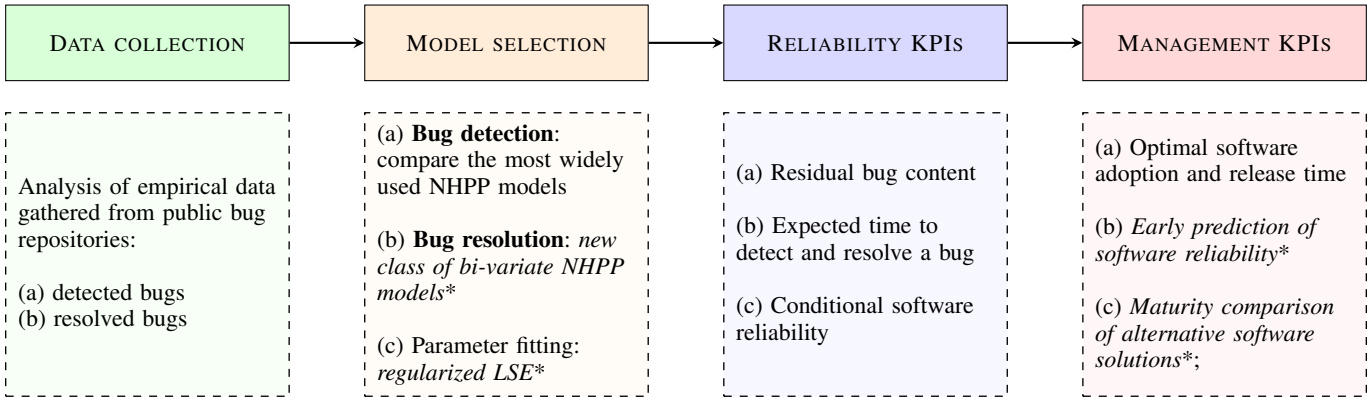


Fig. 1: Assessment of software maturity with Software Reliability Growth Models (SRGM) consists of four steps: (i) data collection, (ii) model selection, (iii) evaluation of reliability KPIs and (iv) evaluation of management KPIs. The process enhancements and novelty proposed in this article are marked with (*).

Model parameter fitting: The common statistical inference techniques to estimate the parameters of SRGM are Maximum Likelihood Estimation (MLE) and Least Square Estimation (LSE), while historically Method of Moments (MoM), graphical and simulation based approaches were used [11]. While MLE is convinient for estimating the confidence intervals, LSE is faster and easier apply to the regularized models described in the following section. Fitting of the model parameters to the empirical data is done either with proprietary general purpose statistical packages, such as SPSS, or specialized tools, such as CASRE [31], SREPT [32] and CARATS [33], just to name the few. In order to account for the newly proposed models, and enhancements in the parameter fitting procedure we have developed our own tool based on the libraries provided by the Python scientific package [34].

Applications of software reliability assessment: Software reliability metrics, such as expected bug detection rate, can be used as a driver to balance the cost of testing and the cost of fixing the bugs during the warranty period, and determine the optimal software release time, which is known as the optimal software release problem. Since the first study by Okumoto and Goel [35], many researches have analyzed the optimal software release problem under different constraints [36]–[42]. Koch et al. [36] provide a cost-benefit analysis for releasing the software after the scheduled deadline, while Yamada et al. [37] propose optimal software release policies minimizing the total expected cost, under minimum reliability requirements. The authors in [39] considered the optimization of the test-effort allocation to different software modules under the constrained budget for the testing expenditures, while Huang et al. [41] analyzed the impact of different test effort allocation strategies. Kimura et al. [40] considered different software maintenance models, i.e. warranty policies. Lai et al. [42] extend the cost model to capture the additional effort of documentation and distribution of the software patches. In this article, we describe two novel use cases, namely i) early prediction of software reliability based on the previous software releases and ii) software maturity metrics as a comparison criteria between the alternative software solutions.

III. SOFTWARE RELIABILITY GROWTH MODELS

During the testing and early operational phase of the software lifecycle the faults are detected and removed, which eventually leads to reliability growth. In the past, many SRGM models have been proposed to estimate and predict the software reliability growth. We focus on a particular class of models that describe the fault detection and fault resolution process as *Non-Homogeneous Poisson Process (NHPP)*, due to their widespread use in the literature.

A. Fault detection process as NHPP

We assume that the initial bug content, i.e. number of introduced bugs present in the software before the start of the testing phase, is a random variable N_0 following the Poisson distribution with the mean a:

$$P(N_0 = n) = \frac{a^n}{n!}e^{-a} \tag{1}$$

The probability of detecting a single bug by the time t follows an arbitrary distribution $F_d(t)$. Assuming the bug detection times are independent and identically distributed random variables, the number of detected bugs by the time t is:

$$P(N_d(t) = k|N_0 = n) = \binom{n}{k} F_d(t)^k (1 - F_d(t))^{n-k} \tag{2}$$

The probability of observing exactly k faults by the time t is then described with the equation.

$$\begin{aligned} P(N_d(t) = k) &= \sum_{n=k}^{\infty} P(N_d(t) = k|N_0 = n)P(N_0 = n) \\ &= \frac{[aF_d(t)]^k}{k!}e^{-aF_d(t)} \end{aligned} \tag{3}$$

The process is fully described with the mean value function $m(t)$, which represents the expected number of detected faults by the time t:

$$E[N_d(t)] = m(t) = aF_d(t) \tag{4}$$

From the mean value function of the fault detection process many reliability features of the software can be estimated. The instantaneous bug manifestation, i.e. bug detection rate is:

$$\lambda(t) = \frac{\mathrm{d}\, m(t)}{\mathrm{d}\, t} = a\, f_d(t) \tag{5}$$

Assuming that the number of initially introduced faults in the software is finite $\lim_{t\to\infty} m(t) = a$, the expected number of the undetected faults in the software, i.e. the residual bug content, is defined as:

$$r(t) = E[a - N_d(t)] = a - m(t) \tag{6}$$

The conditional software reliability is defined as the probability of detecting a new fault in the time interval $(t, t+x]$:

$$R(x|t) = e^{-\int_t^{t+x} \lambda(x)\,\mathrm{d}x} = e^{m(t)-m(x+t)} \tag{7}$$

The expected cost of the software consists of the cost of testing $c_t(t)$ in the pre-release phase, and the cost of removing the fault $c_w(t)$ in the operational phase during the warranty period T_w of the software lifecycle. Assuming that the software is released after T time units of testing, the total cost of software maintenance is:

$$C(T) = \int_{t=0}^{T} c_t(t)\mathrm{d}t + \int_{t=T}^{T+T_m} c_w(t)\lambda(t)\mathrm{d}t \tag{8}$$

We compare eight most widely used NHPP models for modelling of the fault detection process: Musa-Logarithmic, Goel-Okumoto Exponential, Generalized Goel-Okumoto, Inflection S-shaped, Delayed S-Shaped, Yamada-Exponential, Gompertz and Logistic, whose mean value function and failure intensity are given in the Table I.

B. Fault resolution process

The fault resolution process consists of two phases, fault detection and fault correction. If we assume that the fault detection and fault correction are independent, the resulting fault resolution process can be written as [25]:

$$f_r(t) = \int_{x=0}^{t} f_d(t-x) f_c(x) dx = [f_d * f_c](t) \tag{9}$$

where $f_d(t)$ and $f_c(t)$ represent densities of the fault detection and fault correction process, respectively. The mean value function of the resulting fault resolution process is then defined as:

$$m_r(t) = aF_r(t) = a\int_{\tau=0}^{t} [f_d * f_c](\tau) d\tau \tag{10}$$

Equation Eq. (10) can be used to generate different SRGMs from arbitrary distributions for the fault resolution process. However, the proposed models so far have been limited to the combinations for which this integral has a closed form solution, e.g. when both fault detection and correction are Goel-Okumoto processes [25], [28].

$$m_r^{go-go}(t) = a\left[1 - \frac{b_1 e^{-b_2 t} - b_2 e^{-b_1 t}}{b_1 - b_2}\right] \tag{11}$$

By replacing the integral in Eq. (10) with its Piecewise Constant Approximation (PCA), we can obtain a numerical approximation for an arbitrary combination of NHPP models, which can be used for the fitting of the fault report data.

$$F_r(t) = \lim_{\Delta x \to 0} \sum_{i=0}^{n=t/\Delta x} [f_d * f_c](i\Delta x)\Delta x \tag{12}$$

In this article, we compare the four combinations of Generalized Goel-Okumoto and Inflection S-shaped models for fault resolution process, which were preselected due to their performance. We use combined Goel-Okumoto Eq.(11) from [28] as a reference.

C. Fitting of the model parameters

The LSE method, which minimizes the squared distance between the observed and expected data, is used for the fitting of the model parameters. Unconstrained problems in model selection phase (Section V), are solved using Levenberg-Marquardt (LM) algorithm. In Section VI-B we provide the bounds on the model parameters, based on the observed parameter trends in the previous releases. The regularized model is solved using the Trusted Region Reflective (TRF) algorithm. Implementation of both methods is provided by Python scientific computing package [34].

Three Goodness of Fit (GoF) measures are used to evaluate the suitability of the models: Mean Square Error (*MSE*), Theil's statistics (*TS*) and coefficient of determination (R^2). *MSE* is used as to select the best model for individual releases, while *TS* is more suitable to compare the goodness of fit across different software releases. R^2 is used to measure which portion of variance in data can be explained by the model. The three GoF metrics are defined as follows:

$$MSE = \frac{1}{k}\sum_{i=1}^{k} \left(m(t_i) - m_{est}(t_i)\right)^2 \tag{13}$$

TABLE I: Fault detection process as Non-Homogeneous Poisson Process (NHPP)

Model	Abbreviation	Shape	Mean value function	Failure intensity
Musa-Okumoto logarithmic [17]	MUSA(Log)	Concave	$m_{mo}(t) = a\ln(1+bt)$	$\lambda_{log}(t) = \frac{ab}{1+bt}$
Goel-Okumoto exponential [18]	GO(Exp)	Concave	$m_{go}(t) = a(1-e^{-bt})$	$\lambda_{go}(t) = abe^{-bt}$
Generalized Goel-Okumoto [11]	GGO	S-shaped	$m_{ggo}(t) = a(1-e^{-bt^c})$	$\lambda_{ggo}(t) = abct^{c-1}e^{-bt^c}$
Ohba's inflection S-shaped [19]	ISS	S-shaped	$m_{iss}(t) = a\frac{1-e^{-bt}}{1+\phi e^{-bt}}$	$\lambda_{iss}(t) = abe^{-bt}\frac{1+\phi}{(1+\phi e^{-bt})^2}$
Yamada delayed S-shaped [20]	DSS	S-shaped	$m_{dss}(t) = a(1-(1+bt)e^{-bt})$	$\lambda_{dss}(t) = ab^2te^{-bt}$
Yamada exponential [21]	YEX	Concave	$m_{yex}(t) = a(1-e^{-r(1-e^{-bt})})$	$\lambda_{yex}(t) = abre^{-bt}e^{-r(1-e^{-bt})}$
Gompertz [24]	GOMP	S-shaped	$m_{gomp}(t) = ak^{b^t}$	$\lambda_{gomp}(t) = a\ln b \ln k\, b^t\, k^{b^t}$
Logistic [11], [20], [23]	LOGIST	S-shaped	$m_{logist}(t) = \frac{a}{1+ke^{-bt}}$	$\lambda_{logist}(t) = \frac{abke^{-bt}}{(1+ke^{-bt})^2}$

$$TS = \sqrt{\frac{\sum_{i=1}^{k} \left(m(t_i) - m_{est}(t_i)\right)^2}{\sum_{i=1}^{k} m(t_i)^2}} * 100\% \quad (14)$$

$$R^2 = 1 - \frac{\sum_{i=1}^{k} \left(m(t_i) - m_{est}(t_i)\right)^2}{\sum_{i=1}^{k} \left(m(t_i) - \overline{m}\right)^2} \quad (15)$$

$$\overline{m} = \frac{1}{k} \sum_{i=1}^{k} m(t_i)$$

where $m(t_i)$ represent the observed data, and $m_{est}(t_i)$ the data estimated by the model, at time instance t_i of the i-th bug report.

IV. Empirical Data Set: ONOS Controller

The analysis of the software reliability described in the previous section requires complete and uncensored bug reports, which are publicly available only for the open source controllers. At present, there are only two production-grade open source SDN controller platforms, ONOS [2] and OpenDaylight [3], both of them supported by the Linux foundation. In this article we focus mainly on the releases of ONOS controller, because of its focus on high-availability, steady feature development through three month release cycles, and the level of detail provided by its issue tracking system. In the Section VI-C we show that the presented SRGM framework can be applied to OpenDaylight platform as well, and compare the maturity of the two SDN controllers.

A. The scope of ONOS controller

The focus of ONOS, i.e. Open Network Operating System, since its inception has been on providing scalability, high availability and carrier-grade performance fulfilling the requirements of large operator networks [43]. The project is supported by the key partners from the telecom and data center operators and network equipment vendors, such as AT&T, Google, Ericsson, Cisco, just to name a few. Overall, more than 300 developers from more than 60 organizations have contributed to its code base. The code is written mostly in Java and contains at the present 743,531 lines of code[2].

B. Release management

New ONOS releases are distributed every quarter, which provides a steady feature development through incremental upgrades of the code base. The three-month release lifecycle starts with the release planning meeting, followed by three months of code development and integration on the master branch. Two weeks before the official release date feature integration is stopped and only bug fixes are allowed. The support, including security patches and fix for the critical defects, is provided for the six months after the official release date. Thirteen releases (named in the alphabetical order by the birds) have been distributed since December 2014, when ONOS code was opened to the public.

C. Issue tracker

The issues associated to every release are reported in the publicly available Jira tracking system[3]. For the purpose of our analysis we are interested in the issues labelled as "Bugs" rather than new feature requests or enhancements. Such bug repositories represent a valuable source of information, as they contain the detailed fault reports from the live deployments in both lab and operational environments. The bug reports contain the information details such as affected versions, bug description and short summary, priority, date of the report creation, date of its resolution (if applicable). The cumulative number of detected and resolved faults reported over time are shown in Fig. 2. It can be observed from the figure that there is a steady increase in the number of bugs, with the jumps being noticeable around the official release dates.

[2]Source:https://www.openhub.net/p/onos

[3]Data retrieved on February 1, 2018 from ONOS issue tracker: https://jira.onosproject.org/

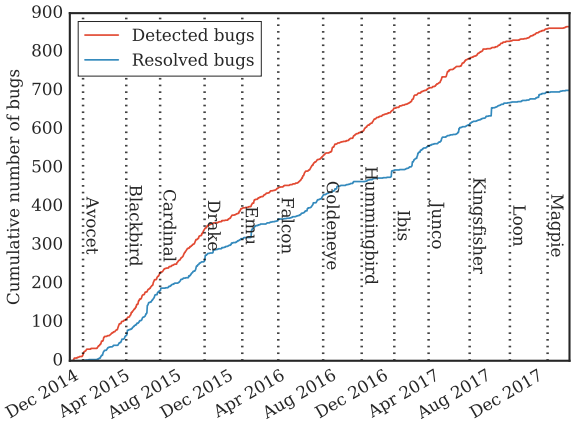


Fig. 2: Number of detected and resolved faults for all releases over time: The official dates of ONOS releases are indicted with the vertical line in the figure.

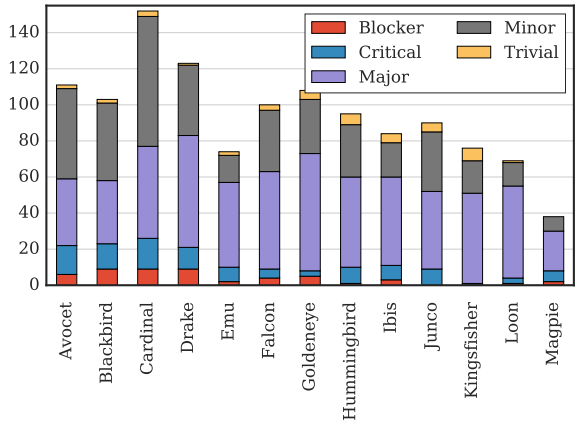


Fig. 3: Number of faults reported for each release, grouped by priority. Only the high impact bugs, i.e. "major", "critical" and "blocker" priority, are included in the analysis.

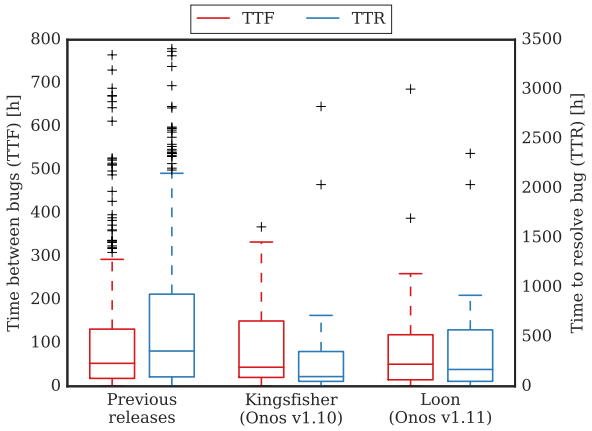


Fig. 4: Distribution of the times between the successive bug reports (TTF), and times to resolve a bug (TTR): comparison between the two latest stable releases with the previous ten releases.

Analysis of the software maturity presented in the previous section assumes that the only changes in the code are due to the bug fixes, and hence, we separate the bugs reports based on the "affected release version" field. The number of the bugs reported for every release, grouped by the priority, are presented in Fig. 3. Note that due to the time overlap between the support periods some of the fault reports affected more than one release. In the analysis of software maturity, "minor" and "trivial" bugs (e.g. loading of the GUI too slow) are ignored, as they do not have an impact on the critical controller operations and often remain unresolved.

D. Data statistics

The latest release, Magpie (Onos v.1.12) was distributed recently and does not have enough samples, i.e. bug reports, for the statistical analysis. Hence, we focus on Kingsfisher (Onos v.1.10), the most recent release whose support cycle has ended, and Loon (Onos v.1.11), and refer to them as the two latest stable releases. We have compared the distributions of the times between bugs (TTF) and the times to resolve the bug (TTR) for Kingsfisher and Loon with the previous ONOS releases, as presented in Fig. 4. The median TTF around 48 h, or two only two days, is consistent for all three data sets. The median TTR showed higher variation, between 168 h to 180 h, or around a week.

Both TTF and TTR show the characteristics of long tail distributions, which makes it difficult for the software management team to estimate, e.g., the effect of the extended testing effort on the improvement of the software quality. The SRGM models, presented in the Section III, add the time dimension to these distributions, and can estimate the parameters, such as the expected number of bugs to be detected in a given time period, with much higher precision.

V. Model selection

The next step after the data collection is to find the best fitting SRGM model to describe the data. In this section we present the best fitting models for the bug detection and bug resolution processes, and discuss their Goodness of Fit (GoF) metrics to determine how well can the models explain the empirical data.

A. Fault detection process

We compare the most widely used SRGM models for the bug detection process presented in the Table I. The empirical data, i.e. the cumulative number of detected bugs, and the estimations of the two best fitting models are presented in Fig. 5. The models are ranked based on the MSE, as it was the optimization criteria of the parameter fitting procedure (Section III-C), which is also indicated in the figure. Time-axis indicates the relative time (in hours)since the beginning of the testing phase.

The analysis has shown that all 3-parameter S-shaped models, i.e. Generalized Goel-Okumoto, Inflection S-Shaped, Gompertz and logistic, fit the data well. Since the difference in MSE between the these models is rather small, we show the estimated number of bugs for the two best fitting models. The best fitting model in the most of the cases is Gompertz (5 out of 12 releases), followed by Logistic (4) and Generalized Goel-Okumoto (3). Inflection S-shaped model also shows very good GoF results, being the second best fit for most of the releases (9 out of 12 releases).

The three GoF metrics for all the models and the releases are compared in Fig. 6. All GoF indicators show consistent results: the best model to describe the number of detected faults across all releases are 3-parameter S-shaped models, showing very good scores in each metric. The concave models, i.e. Musa-Logaritmic, Goel-Okumoto Exponential and Yamada Exponential, could not explain the data, except for the three releases (Avocet, Falcon and Loon) that experience more concave trend. Delay S-shaped shows slightly worse results, compared to the other S-shaped models. This effect is probably due to the fact that this model has only two parameters to tune, one less than the other S-shaped models.

B. Fault resolution process

Arbitrary combination of NHPP models can be used for fitting of the cumulative number of resolved bugs applying the Eq.(10). Here we present the combinations of S-shaped models: Generalized Goel-Okumoto (GGO) and Inflection S-shaped (ISS). The models are abbreviated as a combination of the initials of detection and resolution NHPP processes. For the sake of comparison we also include the reference model from [28] where both fault detection and resolution are modeled as Goel-Okumoto processes, which is the most widely used model due to the analytical tractability of the distributions for the combined process.

The best fitting model for four representative releases, Avocet, Blackbird, Junco and Loon, are shown in Fig. 7. It can be seen that although the proposed models for the fault resolution process could describe the data for some of the releases, the actual data shows higher deviation from the fitted model, then in the previous case. In the first two cases (Fig. 7a and Fig. 7b) the models have shown a very good fit to

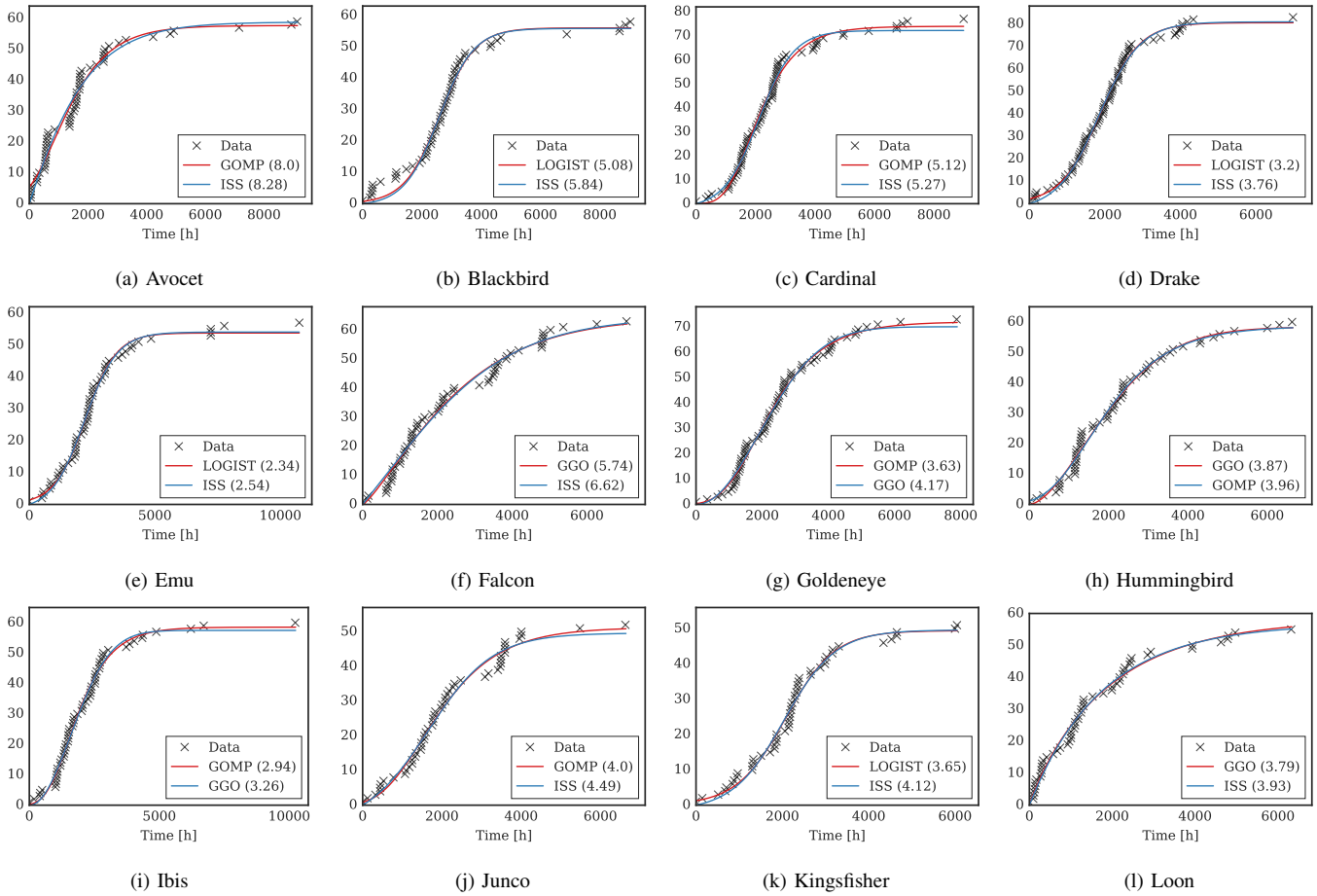
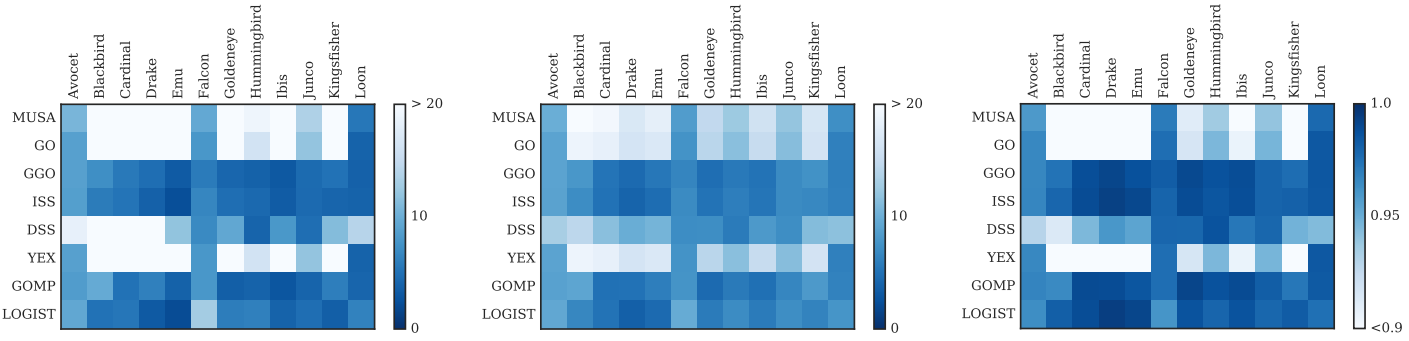


Fig. 5: Comparison of the two best fitting models for bug detection process for all ONOS releases. For most of the releases the best fitting model is Gompertz (5 out of 12 releases), followed by Logistic (4) and Generalized Goel-Okumoto (3). Inflection S-shaped model also shows very good GoF fit, being the second best for most of the releases (9).

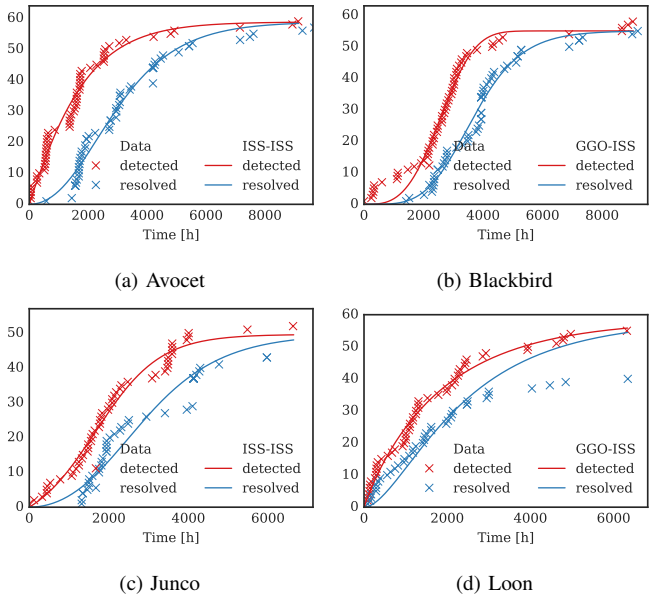


(a) Mean Square Error (MSE) (b) Theil's Statistics (TS) (c) Coefficient of determination (R^2)

Fig. 6: All GoF indicators show consistent results (the darker means better): the best model to describe the number of detected faults across all releases are 3-parameter S-shaped models: Gompertz, Logistic, Generalized Goel-Okumoto and Inflection S-shaped. Two-parameter S-shaped model (Delay S-shaped) shows slightly worse performance, while concave models Musa-Logaritmic, Goel-Okumoto Exponential and Yamada Exponential could not explain the data.

the data. The best fitting models are ISS-ISS and ISS-GGO, respectively.

The two other releases have experienced sudden trend changes around the official release date. In the case of Junco (Fig. 7c) two sudden increases can be detected: the fist one happens around its official release and the second one shortly before the distribution of the subsequent release. Similar behaviour can be observed in several other releases (Goldeneye, Hummingbird, Ibis). Such sudden trend changes due to external signals cannot be captured by the simple combination of NHPP models. The trend shifts due to the changes in the debugging effort shortly before the new upcoming release can



(a) Avocet (b) Blackbird

(c) Junco (d) Loon

Fig. 7: Comparison of the best fitting models for fault resolution process: the proposed models could describe the data for some of the releases, although the actual data shows higher deviation from the fitted model due to the sudden trend changes that require more complex model and more information about debugging effort.

be modelled by introducing the (time) change points in the underlying NHPP models, as described in [44]. This approach requires, the time change points to be provided either manually or defined as additional unknown parameters of the model. In the first approach the generalizability of the model is poor, while in the second approach the estimation of the parameters in the small data sets might be noisy (fitting the model with five or more parameters to dataset with less than 30 examples).

In the case of Loon (Fig. 7d), the trend after the official release is changed, indicating the change in the debugging strategy. Similar behaviour can be observed in (Cardinal, Drake, Emu, Falcon). It has to be noted that in the open source software, such as ONOS, all the users are at the same time the testers, as anybody can report the bug in the public issue tracker. However, only a limited group of people will work on actually fixing the bugs. When this discrepancy between the "test" and "debug" team is too large, or when there is a sudden change in the size of debugging effort, the time scales have to be adjusted accordingly. The models, such as [45], can capture the changes in the test effort, but have the same problem of the accuracy of the parameter fitting on comparatively small data sets.

The same pattern can be observed also in the Fig. 8, where the MSE metrics of the five proposed models for all releases are compared. We observe that ISS-GGO and GGO-ISS outperformed the reference GO-GO model, for all the releases, where fitting was possible.

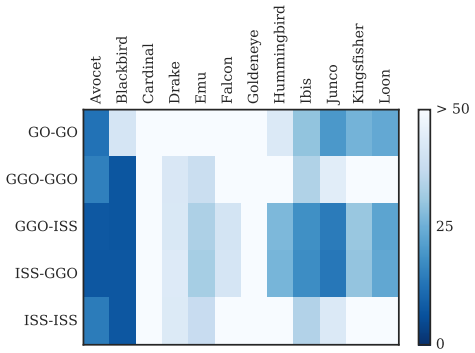


Fig. 8: Comparison of MSE of SRGM models for the fault resolution process: ISS-GGO and GGO-ISS outperformed the reference GO-GO model, for all the releases, for which the fitting was possible.

VI. Applicability of Software Maturity Assessment

In this section we present the software maturity assessment for the three software management problems. First we show how to estimate the optimal software release and software adoption time, based on the reliability and cost criteria, which is a typical use case of SRGM found in the literature [36]–[42]. Then we present two novel use cases, relevant for the SDN community. We show how SRGM parameters can be used for (i) an early estimation of software reliability, and (ii) as criteria to discriminate between alternative controller platforms, e.g. ONOS and OpenDaylight, when reliability has the highest priority.

A. Optimal software release and software adoption time

SDN controllers comprise all the functionalities of the network operating system, and require constant updates to keep up with the velocity of the evolution of the user requirements [4]. In this section we discuss how SRGM can be used to to estimate the quality of the controller software to determine the optimal software release and software adoption time, based on the software reliability and the cost.

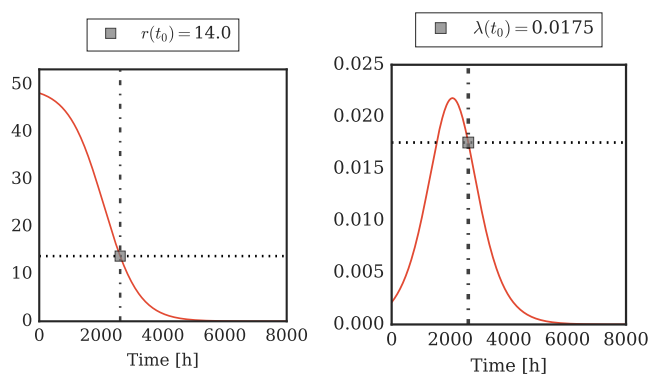
1) Software reliability criteria: Software reliability, defined in the literature as the probability of failure-free software operation for a specified period of time in a specified environment, is an important indicator of software quality. Once the best model to describe the fault report data is selected and the parameters are estimated, it can be used to predict several software reliability parameters: residual bug content, instantaneous fault intensity, conditional software reliability and expected cost, as defined in Section III-A by Eq.(5)-(7).

The software reliability metric for the Kingsfisher release are presented in Fig. 9. Kingsfisher is the most recent ONOS release whose support cycle has ended, and its best fitting model is Logistic. The official release date t_0 is indicated with the vertical line in the figure, and the time is expressed as the relative time since the start of the testing. Note that only severe bugs (bugs with major, critical and blocker priority) are considered.

Residual bug content represents the number of undetected faults remaining in the software. It can be seen in Fig. 9a that the residual bug content was relatively high, as 14 severe bugs were still remaining in the software on the day of its official release. Already three months after the official release, this number has dropped significantly.

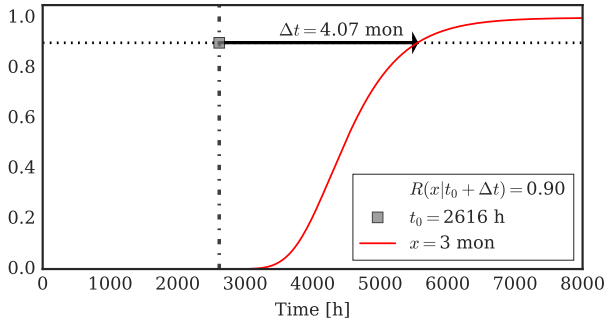
Instantaneous fault intensity, or alternately expected time until the next software failure, can be derived from the parameters of the mean value function. The expected fault intensity, illustrated in Fig 9b, on the day of Kingsfisher's release was at the level of $0.0175h^{-1}$, or equivalent to approximately 2.38 days between detection of successive severe faults. The fault intensity is highly relevant for the software developers, as it can indicate when is the software ready for the release. This metric could help the developers estimate the efficiency of the gains of the additional testing effort.

Conditional software reliability represents the probability of encountering a severe software failure in the time interval $[t, t+x)$. We observe the interval starting with the software adoption time t for a duration x, specified by the user. We show that in order to achieve reliability of $R(x|t) = 0.90$, during maintenance interval of $x = 3$ months, the user should defer the software adoption more than $\Delta t \geq 4$ months after its official release t_0, as illustrated in Fig. 9c.



(a) Residual bug content $r(t)$

(b) Failure intensity $\lambda(t)$



(c) Software reliability $R(x|t)$

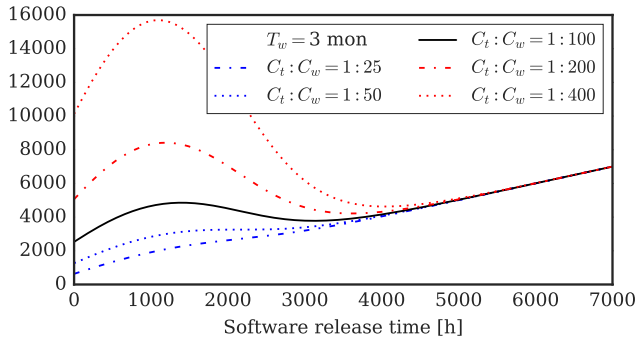
Fig. 9: An example of the optimal software adoption and release time based on the reliability criteria: *residual bug content*, *failure intensity* and *conditional software reliability*. Vertical lines indicate the date of the official Kingsfisher release time (t_0).

2) Software cost criteria: Software management team needs to balance the effort spent on the testing in the pre-release phase, and effort spent on the bug removal of the software in the operational phase. Open source SDN controllers, such as ONOS and OpenDaylight, come with no guarantees provided on the either performance or reliability. However, many commercial solutions provided by network vendors, such as Ericsson and Huawei, are built on top of these controllers.

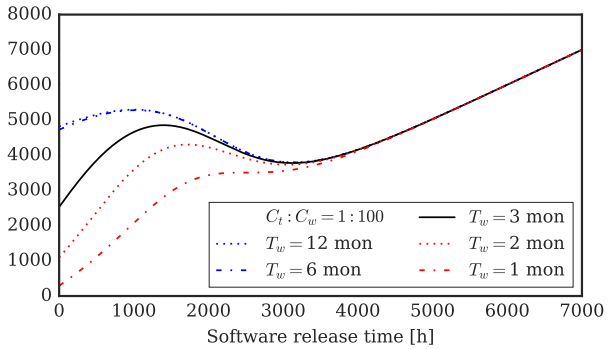
The software cost model, defined by Eq.(8), generalizes the most of the cost models proposed in the literature. The testing cost $c_t(t)$ function accounts for the cost of the software testing team, the cost of the bug removal, the setup and the maintenance of the testing environment, code documentation, etc. The cost during the warranty period $c_w(t)$ includes the penalty paid for every severe outage encountered during the normal operation, the cost of network service interruption, the cost of the bug removal and the support team and sometimes also a discounted value of money for the long support periods. These cost factors must be determined per use case bases. Here, we consider the constant cost functions $c_t(t) = C_t$ and $c_w(t) = C_w$. The software cost function then becomes:

$$C(T) = C_t T + C_w[m(T + T_w) - m(T)] \qquad (16)$$

where $m(t)$ is a mean value function of the best fitting model,



(a) Impact of the relative cost of testing (c_t) and the cost of fixing the code in the maintenance phase (c_w)



(b) Impact of the duration of the maintenance period (T_w) on the total estimated cost.

Fig. 10: Optimal software adoption and release time based on the cost criteria. Software cost as a function of the software release time (T) and the maintenance period (T_w), and relative cost of bug removal in test (C_t) and operational phase (C_w).

discussed in the previous section. Optimal software release time T is obtained by finding the minimum of expected cost function. For the simpler models, e.g. Goel-Okumoto, the optimal solution, i.e. the minimum of the cost function $\frac{\mathrm{d}C(T)}{\mathrm{d}T} = 0$, can be found analytically, while in other cases it has to be found numerically.

In the baseline scenario we assume the relative cost (in unnamed cost units *CU*) of $C_t : C_w = 1[\frac{\text{CU}}{\text{h}}] : 100[\frac{\text{CU}}{\text{bug}}]$ and the warranty period of T_w of 3 months. The impact of different $C_t : C_w$ and T_w on the software cost is illustrated in Fig. 10. In some scenarios the cost function has no clear minimum. In the cases when the cost post-release bug removal is expected to be low, either due to low penalties (Fig. 10a) or the very short warranty period (Fig. 10b), the optimal software release policy is to distribute the software immediately. In the baseline scenario, a clear minimum for the software release T can be observed, which is approximately 40 days after the official software release date ($t_0 = 2616h$), highlighting the cost-benefits of the extended period .

B. Early prediction of software reliability

In order to estimate the SRGM parameters, a large number of samples, i.e. bug reports, has to be provided. In case of ONOS data set, the standard parameter fitting techniques cannot accurately predict the model parameters before 90% of all bug reports are available, which happens for ONOS approximately after six months of testing when it is already too late for software developers (as the software is already released) and the SDN-network operators (since new release is already available). Estimating the SRGM parameters when only few data samples are available is especially difficult for S-shaped models, since they change the concavity somewhere around three months after the start of the testing (See Fig. 5). However, we have noted that the SRGM parameters show very small variation across the releases, thanks to the incremental development strategy of ONOS, as it can be seen in the case of the Gompertz model in Fig. 11. We leverage this fact to guide the parameter fitting procedure, and regularize the model which improves the prediction accuracy in the early phase. The regularization of the model is implemented by restricting the parameter search space, as described in the Section III-C.

The trend observed in Fig. 11 shows several interesting points and hints how the regularization of the search space could be done. The scale parameter a and the shape parameter b show small variations between the consecutive releases. The parameter a varies between 54 and 85; parameter b is in the range (0.99879, 0.99935). The parameter k for the releases that have pronounced S-shape in the range (0,0.02), while for the releases show more concave trend (Avocet, Falcon and Loon) is higher, in the range of (0.5,0.85).

We have explored several parameter regularization strategies. In our previous work we proposed the strategy based on the extreme values, where the search space of every parameter ξ is bounded to [0.9 ξ_{min}, 1.1 ξ_{max}], which represents the range of previously observed parameters extended by 10%. Here we consider the strategy based on the mean m_ξ and the variance σ^2_ξ, where the parameter search space is bounded to

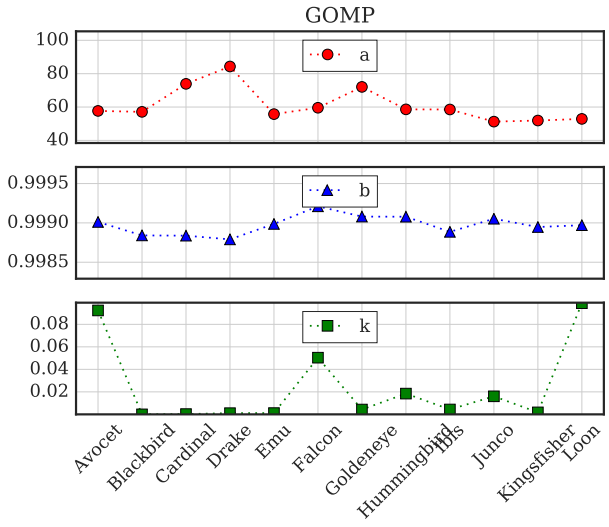


Fig. 11: Estimated parameters of Gompertz model for all releases. The parameters show very small variation across the releases, which allows us to regularize the model for the subsequent releases.

$m_\xi \pm 2\sigma_\xi$. In addition to these two strategies based on the distributions of the parameters, we have considered a strategy based on the trend. We consider an exponentially weighted moving average, defined as:

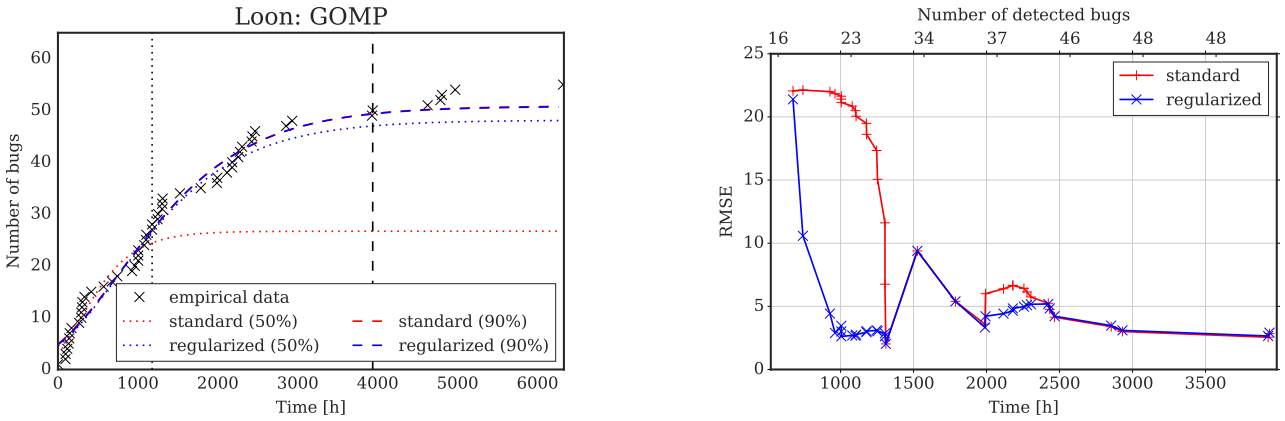
$$m^i_\xi \leftarrow \omega\, \xi^i + (1-\omega)\, m^{i-1}_\xi \tag{17}$$

where the average value of the parameter ξ after i releases m^i_ξ is computed as a weighted sum of the estimated parameter for the i-th release ξ^i and the previous average value m^{i-1}_ξ. Here we assume the $\omega = 0.5$, and bound the parameter search space to $m^i_\xi \pm 2\sigma_\xi$. Note that in cases where the lower bound is negative, the values are capped to zero, due to the nature of the model. The parameter search space bounds with different preparation strategies are compared in Table II.

TABLE II: Gompertz model regularization with parameter prediction strategies, based on: i) extreme parameter values, ii) mean and variance and iii) moving average.

ξ	$[0.9\,\xi_{min}, 1.1\,\xi_{max}]$	$[m_\xi \pm 2\sigma_\xi]$	$[m^i_\xi \pm 2\sigma_\xi]$
a	[50.74, 85.15]	[41.62, 80.65]	[34.21, 73.64]
b	[0.9888, 1.0092]	[0.9987, 0.9992]	[0.9982, 0.9987]
k	[8.2 e-7, 0.0933]	[0.0, 0.0936]	[0.0, 0.0629]

All prediction strategies narrow down the parameter search space: while the first strategy covers extreme values, the range for the other two is more narrow. Overall, all prediction strategies showed improvement over the standard fitting techniques, demonstrating the positive impact of the prior knowledge on the parameter fitting accuracy. The prediction strategy based on the trend shows the unstable performance when parameter experienced the sudden trend changes, as in case of parameter k for Loon release, illustrated in Fig. 11. It might be possible to use a different regularization strategy for each parameter. We leave it for the future work to study the performance of such



(a) Early prediction of mean value function $m_{gomp}(t)$, when limited number of samples are available.

(b) Evolution of Root Mean Square Error (RMSE) with the number of training samples.

Fig. 12: Early prediction of software reliability, when only few samples, i.e. bug reports, are available for the fitting of SRGM parameters. Benefits of regularization can be seen in the evolution of RMSE and mean value function.

hybrid strategies, when more software releases are available and behavioural patterns of each parameter can be estimated more precisely.

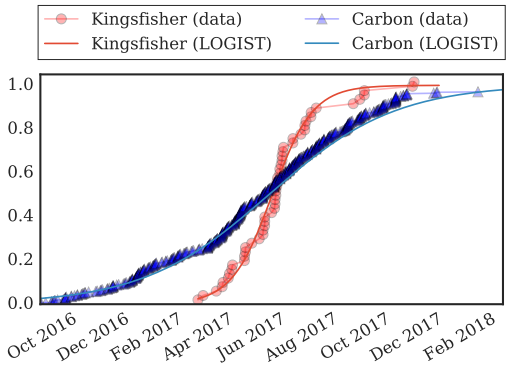
The benefits of regularization on the early prediction of software reliability, can be quantified by observing the estimated mean value function $m_{gomp}(t)$ and the evolution of root mean square error (*RMSE*), when the limited number of the samples, i.e. bug reports are fed to the parameter fitting function. The results for the prediction strategy based on the observed mean and variance are shown in Fig. 12. The impact of the error in parameter estimation is illustrated in Fig. 12a. The error of the estimation with 50% of the available samples with standard fitting techniques is much larger due to the local variations of early samples. It can be observed in Fig. 12b that the regularized model was able to estimate the parameters with higher accuracy and 20% less samples.

In this section we present the Gompertz model, which has the best performance across all releases, being the best fit for five releases, and showing very good results for the other seven. Moreover, the parameters of Gompertz model have shown the smallest coefficient of variation (variance/mean). However, the general conclusions hold as well for the other three 3-parameter S-shaped models. While studying the impact of the model selection, we observe that, in general, the regularization improves the predictive capabilities of SRGM in the early phase of the software lifecycle for all 3-parameter S-shaped model, but the magnitude of the improvement depends on the data set. For Junco release, none of the combinations of the models and prediction strategies show significant improvements with 50% of the samples. This is probably due to the timing of the burstiness of bug reports at the beginning of testing (see Fig. 5). Further improvements could be achieved with smoothing techniques and grouping of the data, e.g, by reducing the time resolution of the bug reports from hours to days or weeks. The limitations of SRGM are further discussed in Section VI-D.

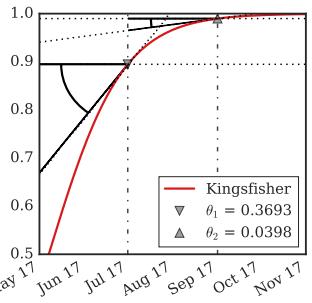
C. Comparison of two SDN controller software solutions: ONOS vs. OpenDaylight

As SDN is gaining the popularity and picking up the momentum, a multitude of commercial and open source SDN controllers have been developed. While the most of the early open source solutions have remained in the research community at the level of the prototype, two projects have been singled out and managed to reach production grade readiness, ONOS and OpenDaylight. OpenDaylight is much larger and older project (see Table **??**), foreseen from the beginning to be the Linux of the networks, supporting a variety of southbound protocols to ensure the smooth transition from the legacy networks. Majority of the OpenDaylight key partners are vendors, and the focus at the beginning was on the the applications in data centers and coexistence with network virtualization technologies, as opposed to ONOS whose primary focus in early days was fulfilling the requirements of service providers. Although the difference in the support of some of the advanced features is still present (e.g. the OpenDaylight support for the wireless networking), the two controller platforms are converging and it is not clear for the network operators which solution to choose. For instance, the commercial SDN controller platform by Ericsson is based on OpenDaylight, while Huawei Agile controller solution is based on ONOS, and AT&T deploys both platforms in its production networks. In this section we address the problem that a network operator might face when it has to choose the optimal SDN controller platform for its network, or alternatively an open source platform as a code base to build his customized controller upon, when code maturity is the major concern.

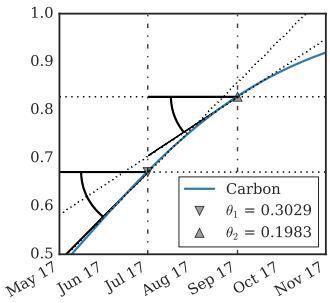
Comparison of the relevant characteristics, e.g. code size and fault density, of OpenDaylight and ONOS controller platforms is presented in Table **??**. We compare Carbon (OpenDaylight v.0.6) and Kingsfisher (ONOS v.1.10) releases, as both of them were distributed approximately at the same time (June 5, 2017 and May 25, 2017, respectively) and sufficient time has elapsed for both controllers to reach the stable



(a) Emprical and fitted data for the two controllers.



(b) Software maturity of Kingsfisher: one (θ_1) and three months (θ_2) after the software release.



(c) Software maturity of Carbon: one (θ_1) and three months (θ_2) after the software release.

Fig. 13: Software maturity evolution over time.

phase. We highlight here the major differences between the two controller platforms, relevant for the analysis of software maturity. First, the release management cycles of the two controllers are different: while ONOS distributes the code in the regular three-month cycles, the lifecycle of OpenDaylight releases is irregular, on average being six months. Moreover, the two controllers use different classification schemes for the importance of the bug reports. ONOS has five well defined categories, OpenDaylight has six, with majority of the bugs (68%) belonging to default "normal" category. Thus, we include the bugs of all priorities in the fault density figure. The two controller platforms had a different approach to their issue tracking systems. While ONOS has been using Jira since its inception for the documentation and management of its bug repository, OpenDaylight relied at the very beginning on the internal mailing list and excel sheets, then used Bugzilla issue tracker in the first 6 releases, and has recently migrated to Jira. Although both issue tracking systems offer the same reporting capabilities, we have found that ONOS bug reports provided higher level of detail and less ambiguity in its bug reports. We observe that the fault density, i.e the number of the bugs detected during the software lifecycle per lines of code, of the two controllers is close to 0.1 $[\frac{\text{bugs}}{\text{kLOC}}]$, with ONOS having slightly lower fault density.

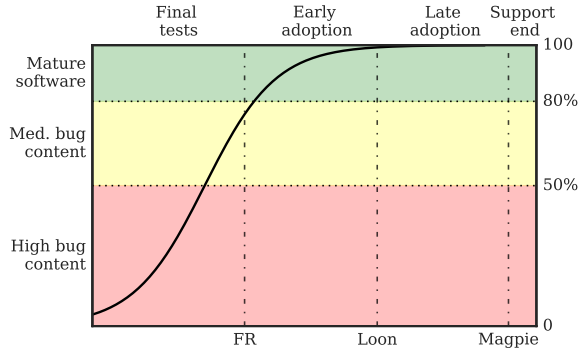
TABLE III: OpenDaylight vs. ONOS.

Controller	OpenDaylight	ONOS
Started	February, 2013	December, 2014
Releases	7	13
Active developers	374	168
No. commits	88,102	11,749
Lines of Code (LOC)	3,860,347	743,531
Reported bugs	493 (Carbon)	76 (Kingsfisher)
Fault density $[\frac{\text{bugs}}{\text{kLOC}}]$	0.128	0.102

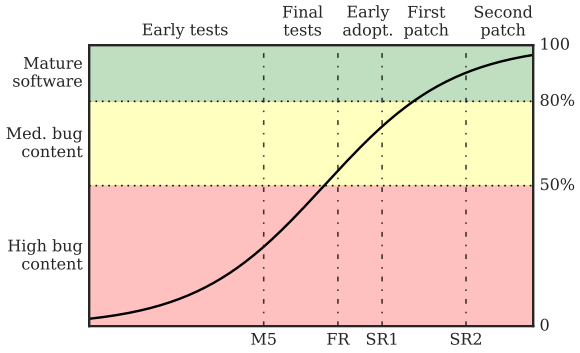
Fault density is a static measure of the code quality, which can be reliably computed only after the software lifecycle is over and the support has ended. Several methods have been proposed for an early estimation of fault density, based on the complexity, programming languages and other software features, which might not always be available to the public. On the other hand, SRGM framework treats the software component as a black box, and provides the estimation of the software reliability, without requiring the information about the code internals. The challenge of direct comparison based on the empirical data between the two releases is illustrated in Fig. 13a. It can be observed that the direct comparison of the empirical data is not straightforward (we assume the bug detection is a realization of the stochastic process), and that on June 1, 2017, both controllers had detected around half of the number of bugs. Instead, it is much more accurate to compare the fitted curves of the two controllers.

In order to compare the reliability and code stability of the two SDN controllers, we propose *software maturity metric*. The software maturity metric is derived from the respective SRGM as $\lambda(t)/m_{max}$, which provides a measure on how far from the stable region (i.e. how close to horizontal line) is the controller software at any given moment. The practical value of our proposed software maturity metric is illustrated in Fig. 13b and Fig. 13c, where the software maturity after one one (θ_1) and three months (θ_2) after the official software release is indicated. The units are expressed as the percentage of detected bugs per day, where zero indicates the stable software. We observe that the maturity of the Kingsfisher improves much faster $\theta_1 = 0.3693[\frac{\%}{\text{day}}] \rightarrow \theta_2 = 0.0398[\frac{\%}{\text{day}}]$, compared to the Carbon $\theta_1 = 0.3029[\frac{\%}{\text{day}}] \rightarrow \theta_2 = 0.1983[\frac{\%}{\text{day}}]$), thanks to the shorter release lifecycles of ONOS.

The software maturity metric can be further used to profile the behaviour of the controller, and quantify the improvement of the software quality over different software lifecycle phases, as illustrated in Fig. 14. Comparison of the maturity evolution over time across different releases can be used to track the progress of the software development process and the efficiency of the testing effort on the improvement of software quality. We recognize the challenges of an early estimation of m_{max}, which have to be estimated before the software lifecycle is over. We can exploit the approach presented in Section VI-B for an early prediction of model parameters. Note that in this particular case, more than 50% of the bug reports were available before the official software release dates for both controllers, in which case our approach for an early prediction can estimate the SRGM parameters with the reasonable accuracy.



(a) Kingsfisher lifecycle: the date of the formal release (FR), subsequent release (Loon) and the end of support (after Magpie release) are indicated in the figure.



(b) Carbon lifecycle: the start of integration testing (Milestone M5), the date of the formal release (FR), and the release of software patches (SR1 and SR2) are indicated in the figure.

Fig. 14: Software maturity in different phases of the controller lifecycle.

D. Threats to validity

The framework presented in this paper comes with certain limitations. The first limitation comes from the fault reports, as the results are only as good as the accuracy of the data sets. While doing the data mining we noticed few inconsistencies. SRGM models require the complete uncensored fault reports, in order to accurately estimate the parameters in the model. Since we can neither fully guarantee the accuracy nor the completeness of the reported data in the issue trackers, we do not emphasize the numerical results, but rather focus on the general approach to quantify the software reliability.

The second limitation comes from SRGM models. The models assume independent times between the consecutive fault reports, which is not entirely true since occasionally several related bugs were reported at the same time. The models also assume that every undetected fault contributes the same to the fault manifestation rate. The time in our study represents the calendar time. It would be more accurate to consider the actual test effort in men-hours and CPU time, but this information is not available large open source projects. Although we cannot guarantee that any SDN controller software can be modelled as mixture of simple SRGM models, previous studies have shown that described models can be successfully applied to many large open source software products, such as Apache Web Server, Mozilla Firefox, and Eclipse IDE (see Section II).

VII. Conclusion

In this article we presented a framework to estimate and predict the maturity of SDN controllers, based on Software Reliability Growth Models (SRGM). SRGM are used to model the stochastic behaviour of bug manifestation and correction processes, based on empirical data of previous outages. We analyzed data provided by the public bug repositories of the two biggest open source SDN controller platforms, ONOS and OpenDaylight.

Our study showed that the three-parameter S-shaped models describe well the stochastic behaviour of the bug detection process for all software releases included in our study. On the other hand, the bug correction process, for most new releases, experienced sudden trend changes that cannot be captured well with combination of standard three-parameter models. For the software releases with stable trend, our proposed class of bivariate models for bug correction process outperformed the reference model.

We have demonstrated how software reliability metrics, derived from SRGM, can be used to guide software developers and network operators to decide when the software is mature enough to be released and deployed in an operational environment. We also proposed two novel applications of SRGM relevant for developers of SDN controllers and operators of SDN-based networks. We proposed model regularization techniques for an early prediction of software reliability based on the observed trend of model parameters of previous software releases. We also define a new software maturity metrics, which can be used as a selection criteria for controller candidates, e.g. ONOS and OpenDaylight, and measure the testing progress and software quality.

References

[1] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[2] ON.Lab, "ONOS: Open Neetwork Operating System," 2017. [Online]. Available: http://onosproject.org/

[3] Linux Foundation, "Opendaylight," 2017. [Online]. Available: https://www.opendaylight.org/

[4] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from Google's network infrastructure," in *Proceedings of ACM SIGCOMM Conference*. ACM, 2016, pp. 58–72.

[5] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "Crystalnet: Faithfully emulating large production networks," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 599–613.

[6] J. Vestin, A. Kassler, and J. Akerberg, "Resilient software defined networking for industrial control networks," in *10th International Conference on Information, Communications and Signal Processing (ICICS)*. IEEE, 2015, pp. 1–5.

[7] E. Sakic and W. Kellerer, "Response time and availability study of RAFT consensus in distributed SDN control plane," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 304–318, 2017.

[8] P. Vizarreta, P. Heegaard, B. Helvik, W. Kellerer, and C. Mas Machuca, "Characterization of failure dynamics in SDN controllers," in *9th International Workshop on Resilient Networks Design and Modeling (RNDM)*. IEEE, 2017, pp. 1–7.

[9] P. Vizarreta, K. Trivedi, B. Helvik, P. Heegaard, W. Kellerer, and C. Mas Machuca, "An empirical study of software reliability in SDN controllers," in *13th International Conference on Network and Service Management (CNSM)*. IEEE, 2017, pp. 1–9.

[10] ETSI GS NFV-REL 003 V1.1.1 (2016-04), *Network Functions Virtualisation (NFV);Reliability; Report on Models and Features for End-to-End Reliability*, 2016.

[11] M. R. Lyu *et al.*, *Handbook of software reliability engineering*. IEEE computer society press CA, 1996.

[12] Y. Zhou and J. Davis, "Open source software reliability model: an empirical approach," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–6.

[13] C. Rahmani, H. Siy, and A. Azadmanesh, "An experimental analysis of open source software reliability," *Department of Defense/Air Force Office of Scientific Research*, 2009.

[14] B. Rossi, B. Russo, and G. Succi, "Modelling failures occurrences of open source software with reliability growth," in *IFIP International Conference on Open Source Systems*. Springer, 2010, pp. 268–280.

[15] S. M. Syed-Mohamad and T. McBride, "Reliability growth of open source software using defect analysis," in *International Conference on Computer Science and Software Engineering*, vol. 2. IEEE, 2008, pp. 662–667.

[16] N. Ullah, M. Morisio, and A. Vetro, "A comparative analysis of software reliability growth models using defects data of closed and open source software," in *35th Annual IEEE Software Engineering Workshop (SEW)*. IEEE, 2012, pp. 187–192.

[17] J. D. Musa and K. Okumoto, "A logarithmic poisson execution time model for software reliability measurement," in *Proceedings of the 7th international conference on Software Engineering*. IEEE Press, 1984, pp. 230–238.

[18] A. Goel and K. Okumoto, "Time-dependent error detection rate model for software and other performance measure," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, p. 285, 1985.

[19] M. Ohba, "Software reliability analysis models," *IBM Journal of research and Development*, vol. 28, no. 4, pp. 428–443, 1984.

[20] S. Yamada, M. Ohba, and S. Osaki, "S-shaped reliability growth modeling for software error detection," *IEEE Transactions on Reliability*, vol. 32, no. 5, pp. 475–484, 1983.

[21] S. Yamada, H. Ohtera, and H. Narihisa, "Software reliability growth models with testing-effort," *IEEE Transactions on Reliability*, vol. 35, no. 1, pp. 19–23, 1986.

[22] S. S. Gokhale and K. S. Trivedi, "Log-logistic software reliability growth model," in *3rd IEEE International High-Assurance Systems Engineering Symposium*. IEEE, 1998, pp. 34–41.

[23] C.-Y. Huang, M. R. Lyu, and S.-Y. Kuo, "A unified scheme of some nonhomogenous poisson process models for software reliability estimation," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 261–269, 2003.

[24] K. Ohishi, H. Okamura, and T. Dohi, "Gompertz software reliability model: Estimation algorithm and empirical validation," *Journal of Systems and Software*, vol. 82, no. 3, pp. 535–543, 2009.

[25] Y. Wu, Q. Hu, M. Xie, and S. H. Ng, "Modeling and analysis of software fault detection and correction process by considering time dependency," *IEEE Transactions on Reliability*, vol. 56, no. 4, pp. 629–642, 2007.

[26] H. Pham, L. Nordmann, and Z. Zhang, "A general imperfect-software-debugging model with s-shaped fault-detection rate," *IEEE Transactions on Reliability*, vol. 48, no. 2, pp. 169–175, 1999.

[27] C.-Y. Huang and M. R. Lyu, "Estimation and analysis of some generalized multiple change-point software reliability models," *IEEE Transactions on Reliability*, vol. 60, no. 2, pp. 498–514, 2011.

[28] P. Kapur, H. Pham, S. Anand, and K. Yadav, "A unified approach for developing software reliability growth models in the presence of imperfect debugging and error generation," *IEEE Transactions on Reliability*, vol. 60, no. 1, pp. 331–340, 2011.

[29] S. S. Gokhale, M. R. Lyu, and K. S. Trivedi, "Analysis of software fault removal policies using a non-homogeneous continuous time Markov chain," *Software Quality Journal*, vol. 12, no. 3, pp. 211–230, 2004.

[30] H. Okamura and T. Dohi, "A generalized bivariate modeling framework of fault detection and correction processes," in *28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 35–45.

[31] M. R. Lyu and A. Nikora, "CASRE: a computer-aided software reliability estimation tool," in *5th International Workshop on Computer-Aided Software Engineering*. IEEE, 1992, pp. 264–275.

[32] K. S. Trivedi, "SREPT: A tool for software reliability estimation and prediction," in *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2002, p. 546.

[33] C.-C. Chen, C.-T. Lin, H.-H. Huang, S.-W. Huang, and C.-Y. Huang, "CARATS: a computer-aided reliability assessment tool for software based on object-oriented design," in *IEEE Region 10 Conference TENCON*. IEEE, 2006, pp. 1–4.

[34] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[35] K. Okumoto and A. L. Goel, "Optimum release time for software systems based on reliability and cost criteria," *Journal of Systems and Software*, vol. 1, no. 4, pp. 315–318, 1980.

[36] H. S. Koch and P. Kubat, "Optimal release time of computer software," *IEEE Transactions on Software Engineering*, no. 3, pp. 323–327, 1983.

[37] S. Yamada and S. Osaki, "Cost-reliability optimal release policies for software systems," *IEEE Transactions on Reliability*, vol. 34, no. 5, pp. 422–424, 1985.

[38] P. Kapur and R. Garg, "Optimal sofware release policies for software reliability growth models under imperfect debugging," *RAIRO-Operations Research*, vol. 24, no. 3, pp. 295–305, 1990.

[39] S. Yamada, T. Ichimori, and M. Nishiwaki, "Optimal allocation policies for testing-resource based on a software reliability growth model," *Mathematical and Computer Modelling*, vol. 22, no. 10-12, pp. 295–301, 1995.

[40] M. Kimura, T. Toyota, and S. Yamada, "Economic analysis of software release problems with warranty cost and reliability requirement," *Reliability Engineering & System Safety*, vol. 66, no. 1, pp. 49–55, 1999.

[41] C.-Y. Huang and M. R. Lyu, "Optimal release time for software systems considering cost, testing-effort, and test efficiency," *IEEE Transactions on Reliability*, vol. 54, no. 4, pp. 583–591, 2005.

[42] R. Lai, M. Garg, P. K. Kapur, and S. Liu, "A study of when to release a software product from the perspective of software Reliability Models," *Journal of Software*, vol. 6, no. 4, pp. 651–661, 2011.

[43] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the 3rd workshop on Hot topics in software defined networking (Hot SDN)*. ACM, 2014, pp. 1–6.

[44] C.-Y. Huang, T.-Y. Hung, and C.-J. Hsu, "Software reliability prediction and analysis using queueing models with multiple change-points," in *3rd IEEE International Conference on Secure Software Integration and Reliability Improvement*. IEEE, 2009, pp. 212–221.

[45] H. Pham and X. Zhang, "Nhpp software reliability and cost models with testing coverage," *European Journal of Operational Research*, vol. 145, no. 2, pp. 443–454, 2003.