On the impact of debugging on software reliability growth analysis: a case study

Marcello Cinque[1], Claudio Gaiani[2], Daniele De Stradis[2], Antonio Pecchia[1], Roberto Pietrantuono[1], and Stefano Russo[1]

[1] [1]Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione, Universita' degli Studi di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy.
{marcello.cinque, antonio.pecchia, roberto.pietrantuono, stefano.russo}@unina.it
[2] Assioma.Net, Via G. Spano, 6/11 - 10134 Torino
claudio.gaiani@assioma.net

Abstract. Reliability is one of the most relevant software quality attributes. A common way to analyze software reliability is the use of software reliability growth models (SRGM). There is a variety of SRGMs that have been proposed across years, which aim at providing estimates of reliability at a given time, as well as predictions of reliability that will be achieved with the advancement of testing. One of the most impacting assumptions of SRGMs is the immediate debugging of detected faults. But in reality, the debugging process is increasingly complex in real project settings, and its impact cannot be neglected at all. This paper reports the results of a case-study in which we analyze the debugging process of a Customer Relationship Management (CRM) system, and study its impact on SRGM-based reliability estimation and prediction.

Keywords: Software reliability, Software reliability growth model, SRGM, reliability prediction, reliability estimation, debug, repair, fault removal, fix

1 Introduction

Software reliability estimation and prediction play a key role for project schedule, cost, and quality control. A substantial field of research, developed over the past three decades, is about the usage of **software reliability growth models** (SRGMs). SRGMs is a wide class of models conceived to fit inter-failure times from test data, in order to estimate the next time to failure based on the observed trend. Information provided by SRGMs can be used to: *i)* estimate software reliability at a given time, as well as the number of residual defects in the software, *ii)* predict the expected reliability given a budget (e.g., in terms of testing time or testing effort), *iii)* schedule the optimal time to release at a given reliability level, *iv)* compare actual and estimated release time in order

to identify delays and their causes. SGRMs make a set of common assumptions to meet the mentioned objectives. Most usual assumptions are immediate debugging, perfect debugging, dependent inter-failure times, equal probability to find a failure across time units [1]. In the literature, a greater attention is being paid to the assumptions about debugging, since their impact is more relevant than the others in real projects. While several works introduce modeling approaches to overcome these assumptions [2], [4], [17], several empirical studies make it evident that debugging is a complex process to model in real-world projects [19], [20], [21]. There are many factors impacting the computation of the actual repair time, and the regularity of the debugging process, such as the type of defect, its priority/severity, and human factors (e.g., skills and attitude of the developer(s) involved in the fixing process and clearness of the bug description).

In these settings, assumptions on debugging are easily violated as complexity and size of a software project increase. Average times to repair a defect are very unlikely to be *immediate*, as repairs are unlikely to be perfect (for example, debug is likely to introduce regression bugs especially in continuous development scenarios and large systems). The *debugging process might even become a bottleneck for project releases, and its impact cannot be neglected at all.*

The impact of an irregular and variable debugging process, besides hampering a correct modeling, influences the assessment of release quality estimation and on SRGM-based predictions. These can determine errors in taking decisions on when to stop testing or on what is the expected reliability and/or residual defectiveness, typical objectives of SRGMs.

In this paper we analyze 3,392 real-world issues of an industrial case-study[3] collected over a time period of two years. On these data, we first *i)* use SRGMs to characterize the software reliability growth under the assumption of immediate debugging actions; *ii)* then, we characterize out the debugging process; *iii)* finally, we evaluate the impact of the debugging time evolution on reliability estimation and prediction, and thus on release scheduling performed by SRGMs. The study shows that:

1. Collected issues are amenable to be modeled by SRGMs; we applied a set of 7 models to fit data and found the truncated logistic and truncated normal SRGM being the best fitting models. Hence, despite the real data do not fulfill classical SRGM assumptions, such as dependent inter-failure times and equal failure detection probability, the models have shown to be robust. However, since the models are built on *opened* issues, nothing can be said about the non-immediate debugging assumption Therefore, such models are useful for SRGMs-based predictions, only provided that the underlying debugging time is negligible.
2. The observed debugging process has a non-negligible time, as expected, equals to 12.8 days on average. The statistical characterization highlights a good quality of the process. Issues are closed regularly and in a reasonable time and, as desirable, severe issues are solved faster than minor ones. It

[3] The actual name of the system in not disclosed here due to confidentiality reasons

also emerged that the queuing time of issues is not negligible (about a third of the overall fixing time), hence it has to be considered to avoid inaccurate results.
3. The non-immediate debugging has an impact on both reliability estimation and prediction, and on the optimal release schedule, in a different way: in both cases the impact is dependent on the debugging process quality in terms of debugging time and debugging time variation, but while the impact on reliability estimation is quite insensitive with respect to the testing time dimension, the release schedule prediction error can greatly variate according to the testing time. As testing times proceeds, the optimal release schedule prediction can be affected considerably by the debugging time.

The paper is organized as follows. Section 2 presents related work in the area of SRGMs, while Section 3 introduces available dataset. Section 2 SRGMs that have been used to model available data. Section 5 discusses the debugging process and the impact of debugging on reliability and best schedule estimation, and Section 6 concludes the work.

2 Related Work

Reliability analysis can be conducted by modeling approaches, by measurements, and by hybrid approaches [5] [6] [7] [23] [24]. Reliability is very tightly related to testing, since the latter is supposed to grow as more testing time is devoted to the software [8]. In the testing phase, one of the most successful approaches for reliability analysis is SRG modeling. There are many SRGMs available in the literature. A very successful one was proposed by Goel and Okumoto in 1979 [9], describing the failing process by an exponential *mvf* distribution. Other common models were proposed later, such as: the generalized version of the exponential model, which uses the Weibull distribution [13]; the S-Shaped model [10], conceived to capture the possible increasing/decreasing behaviour of the failure rate during the testing process; the Gokhale and Trivedi log-logistic model [11], that also follows an increasing/decreasing pattern describing the initial phase of testing as characterized by a slow initial learning phase; more recent ones, as the models based on the Gompertz SRGM, proposed by Ohishi *et al.* [15], derived from the statistical theory of extreme-value. Many other models have been proposed in the literature, and several tools have been developed to deal with parametrization and fitting of models (such as SMERFS, SoRel, PISRAT, and CASRE). All these models are based on a set of common assumptions, the most impacting ones being the immediate and perfect debugging.

In the past, some research has defined SRGMs accounting for the perfect debugging assumption, namely by including the imperfect debugging in the model. For instance, there is a class of SRGMs known as infinite-failure models, which, contrarily to the finite-failure models, assume that an infinite number of faults would be detected in infinite testing time [2]. These are meant to capture debugging where faults may be reintroduced. An example is the Musa-Okumoto

logarithmic Poisson execution time model [3], the more recent failure-size proportional model proposed in [4], as well as the model by Jain *et al.* [16].

As for the *immediate* repair assumption, some researchers modeled the debugging process through queuing models, thus considering also the non-immediate debugging time. For instance, the work in [17] uses a queue model for the correction process, while authors in [18] discuss both finite and infinite server queuing models for reliability measurement through SRGMs. On one side there are these models that can take into account the debugging times from a theoretical point of view; on the other side, there are empirical studies that analyze the characteristics of bug fixing process in real projects, which make it evident that debugging is a complex process to model. Thus, parametrizing models can be a non-trivial task. There are several factors that impact the computation of the actual debugging time. For instance, the study in [19] reports an analysis on 1500 defects revealed in 5 years on an IBM middleware, classifying defects per topic and developer expertise, showing that the time to repair is impacted from these two factors. Zhang et *al.* [20] found some factors influencing the time lag between the defect assignment to a developer and the actual starting of the repair action, through a case-study on 3 open source software. They found that the assigned severity, the bug description, and the number of methods and changes in the code as impacting factors. The work in [21] reports a study specifically focused on finding bottlenecks in the issue management process, through a case-study of the Apache web-server and the Firefox browser. The main cause of inefficiency is the time lag in which the correction is verified after the repairing to confirm the correct resolution. These are all factors that cause irregularities in the debugging process, and can make the impact of debugging on reliability estimation/prediction more variable and difficult to control. In this work, such impact is studied with reference to one specific real-world case-study.

3 Data Source

The target system is a Customer Relationship Management (CRM) software for a multinational company operating in the healthcare sector. The system follows a classical three-layer architecture, with a Frontend, a Backend, and a Database, glued together with an Enterprise Application Integration (EAI) layer. The system provides classical CRM functionalities, such as, sales management, user profiles, agenda, contacts, inventory, procurement of goods, and various reporting tools. Data used in this study are issues collected from the tracker of the target system. In particular, we use data extracted from 3392 issues collected for 30 months, ranging from September 2012 to January 2014. For every issue, several attributes are available, such as:

- *Status*: the working status of the issue; the following statuses are considered (detailed in section 5): new, published, in study, launched, completed, tested, delivered, suspended, and closed;
- *Timestamps*: the dates of every status transition are tracked; the most important timestamps are the ones related to the *new* status (when an issue

is opened) and the *closed* status (when an issue is solved); the timestamps related to intermediate statuses are also useful to analyze the phases of the workflow, such as, how much time an issue is queued waiting to be fixed (e.g., from the *new* to the *in study* status) or how much time is devoted to the fixing itself (e.g., from the *in study* to the *closed* status);
- *Assignees*: the number of resources allocated on the issue;
- *Affected Version*: the version of the system affected by the issue;
- *Severity*: the severity of the issue, classified in blocking, major and minor;
- *Affected Component*: the name of the component (and subcomponent) affected by the issue;
- *Resolution*: final classification of the issue, as *fixed*, *won't fix*, or *not a defect*.

Data have been polished to remove inconstancies and useless issues (for instance, newly opened issues that are canceled without being treated). Finally, 3335 issues have been considered in the study.

4 Reliability analysis through SRGMs

In this Section, we present the analysis conducted of reliability growth *vs.* testing time, starting from the collected issues. To this aim, we adopt software reliability growth models (SRGMs). At this stage, we consider the testing process followed by an immediate and perfect debugging. Thus, the model is obtained by considering the opening time of the issues.

We consider the most common class of SRGMs, those describing failure occurrence as a non-homogeneous Poisson process (NHPP). These are characterized by the parameter of the stochastic process, $\lambda(t)$, indicating the failure intensity, and by the *mean value function* (*mvf*), *m(t)*, that is the expectation of the cumulative number of defects detected at time t [11]: $N(t)$: $m(t) = E[N(t)]$; $\frac{dm(t)}{dt} = \lambda(t)$. The different types of SRGMs can be described by their mean value function, that appears in this form $m(t) = aF(t)$, where a is the expected number of total defects.

A common belief about SRGMs is that it does not exist one single model able to work well with any set of data, but the best model needs to be selected for each specific context. For our purpose, we consider the list reported in Table 1, in order to capture the actual behavior of the testing process. It also reports the corresponding expression of the mean value function (*mvf*); the estimated number of defects is always the *mvf*'s first parameter, a.

Therefore, to select the best fitting SRGM, we try to fit the issue data with every SRGM listed in Table 1 by the EM algorithm [14], and then perform a goodness of fit (GoF) test by means of the Kolmogorov-Smirnov (KS) test. Among the SRGMs with KS test satisfied, we use the *Akaike Information Criterion* (AIC) to select the model, taking the SRGM with the lowest AIC value (as in [14]).

Figure 1 shows the entire set of the raw data and the model fitting them. We can note a pronounced saturation around the day 100, causing no model

Table 1: Software Reliability Growth Models

Model	*m(t) function*
Exponential	$a \cdot (1 - e^{-bt})$
S-shaped	$a \cdot [1 - (1 + gt)e^{-bt}]$
Weibull	$a \cdot (1 - e^{-bt^{\gamma}})$
Log Logistic	$a \cdot \frac{(\lambda t)^{\kappa}}{1+(\lambda t)^{\kappa}}$
Log Normal *	$a \cdot \Phi(\frac{log(t)-\mu}{\sigma})$
Truncated Logistic	$a \cdot \frac{(1-e^{-t/\kappa})}{(1+e^{-(t-\lambda)/\kappa})}$
Truncated Normal *	$a \cdot \frac{\Phi((t-\mu)/\sigma)}{1-\Phi(-\mu/\sigma)}$
* Φ indicates the normal distribution	

satisfying the KS test. For visually capturing the trend, we however reported the model having the lowest AIC value among the SRGMs, which is a truncated logistic one. At a closer look, we noticed that the saturation point around day 100 corresponds to the release of the first major version; data from the day 190 on refer to defects belonging to version 2.0 of the software. Thus, we split the dataset into two groups, according to the release (Figure 5(a)-5(b)).

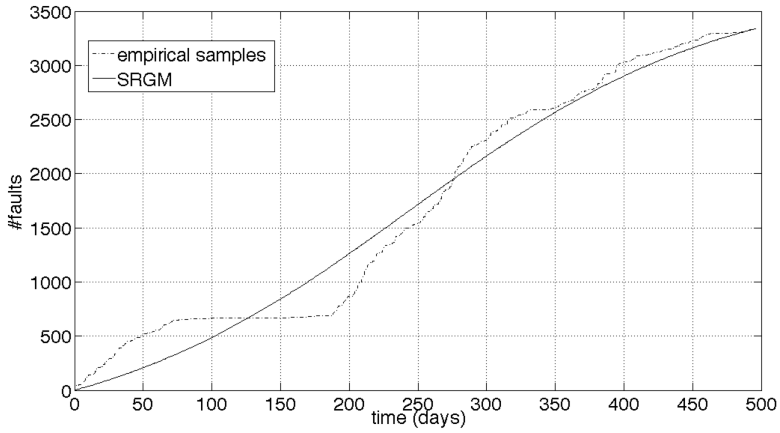
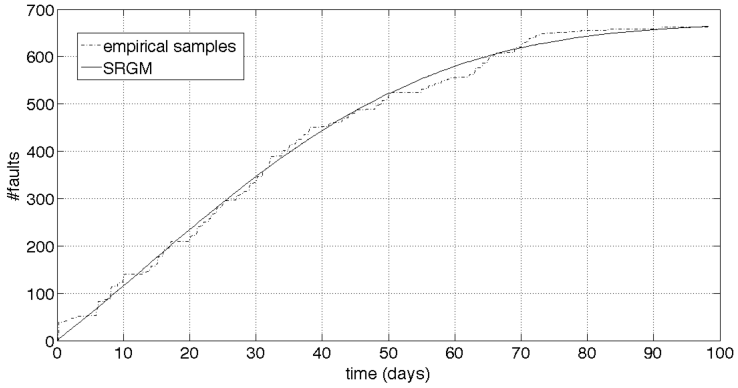
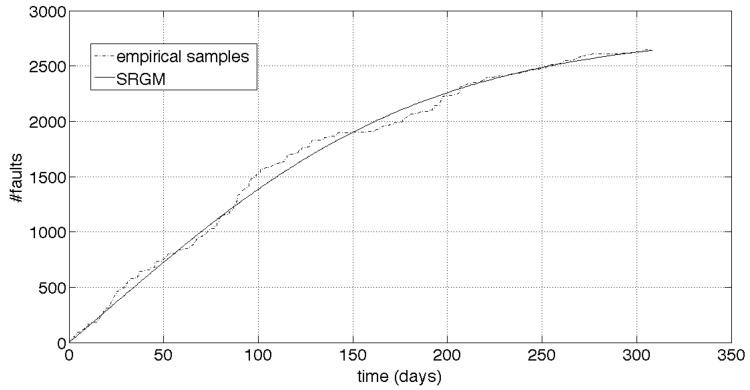


Fig. 1: Cumulative number of opened issues

Table 2 shows the statistics of the selected models for the two versions. The estimates in this case are very close to actual data, both satisfying the KS test. Such models can provide estimates and predictions in terms of: residual issues at a given time, percentage of detected issues over the total expected ones; failure



(a) Cumulative number of opened issues for version 1



(b) Cumulative number of opened issues for version 2

Fig. 2: Cumulative number of opened issues and fitted SRGMs for both versions

Table 2: SRGM Fitting Results

Version	*Current # of Defects*	*Selected SRGM*	*Current Estimate of # of Defects*	*KS Test true at*	*Exp. # of Defects at $t = \infty$*	*Scale param.*	*Shape param.*	*AIC*
1	665	Trunc. Normal	663.93	90%	671.38	13.00	34.78	-1491.77
2	2647	Trunc. Logistic	2640	90%	2808.22	20.95	85.54	-6834.93

intensity; reliability. Note that these measures are equivalent to each other, since the expected cumulative number of issues at time t is the $mvf(t)$ function, whose first derivative is the failure intensity function $\lambda(t)$; the latter can be used in the computation of reliability (e.g., [11], [22]). Considering these measures, testers can evaluate, for instance, what is the best time to release.

Taking the version 1 model, it is evident how the testing process is getting saturated, detecting less and less faults as the testing proceeds. The process detected more than 99% of the total expected defects and will take much time to detect residual ones: thus this has been a good time to release. Looking at version 2, testers detected roughly the 94% of total expected defects. If, for instance, they decide to release with the same quality as version 1, i.e., at 99%, the model predicts a testing time of 448 days, thus still 448 - 308 = 140 days of residual testing days. Based on these and similar analyses, tester can take decisions on when to stop testing.

Such types of analysis have been conducted on the opening time of the issues. This means that 99% of quality is assumed to be the 99% of the total estimated issues that have been *opened*: this is the actual released quality only under the assumption that the correction of those issues have been done immediately and perfectly. In fact the actual quality is given by the *closed* issues, whose fixing contributes to the actual reliability growth. In the next Section, we first analyze the debugging process in our case study, and then we remove the immediate debugging assumption in the SRGMs, in order to see what changes in the reliability analysis.

5 The debugging process

5.1 Debugging process characteristics

In the reality, the debugging process is not perfect and immediate, but it follows the workflow depicted in Figure 3 for the system under analysis. When an issue is opened, it becomes *new* and it is enqueued, waiting to be processed (*published* status). Once an issue starts to be processed (*in study*), it is assigned (*launched*) to a developer and, once *completed*, it could be assigned to another developer for further processing, when needed. Then, the amendment is *tested*, *delivered*, and finally *closed*. It may happen that the testing process may fail. In this case, the issue becomes *suspended* after delivered, and then reopened again for another cycle of processing (transition in the *published* status). From the data, we also found issues that never go in the closed status, either because still under

processing (e.g., this happens for recent issues opened in january 2014) or because finally classified with a "won't fix" resolution.

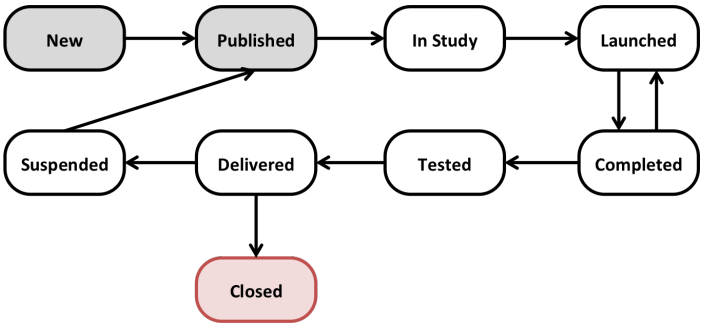


Fig. 3: Workflow of the debugging process. Statuses in gray represent the "idle" part of the process, where issues are enqueued waiting to be processed.

Clearly, the overall debug process is far from being immediate. Just to have an idea, defining TTFix (Time To Fix) as the overall time needed for an issue to transit from the *new* status to the *closed* status, the mean TTFix on the overall dataset is equal to 12.8 days.

More in detail, we have evaluated the following statistics from issues' data:

- $MTTFix$ (in days): the expected value of the TTFix;
- MED_{TTFix} (in days): the median of the TTFix;
- $StdDev_{TTFix}$ (in days): the standard deviation of the TTFix;
- MQT (in days): the Mean Queuing Time, that is, the expected value of the time that an issue is enqueued waiting to be processed;
- MED_{QT} (in days): the median of the queuing time;
- $StdDev_{QT}$ (in days): the standard deviation of the queuing time;
- $\#Res$: average number of resources allocated to the issue;
- *Kurtosis*: it indicates the peakedness of the distribution;
- *Skewness*: it measures the asymmetry of the distribution; a positive value indicates a right-tailed distribution, whereas a negative value indicates a left-tailed one.

The achieved statistics are summarized in Table 3, where issues have been grouped according to their severity (minor, major, and blocking). From the statistics, it can be noted that overall the debug process has a good level of quality. Issues are closed in a reasonable time framework (from 9 to 16 days on average, 5-6 days median), even if very variable (high standard deviation), depending on the complexity of the issue. Moreover, as desirable, blocking issues are solved faster than major issues, that, in turn, are solved faster than minor issues. Another indication of the quality of the process is provided by kurtosis and skewness indicators. The high value of kurtosis denotes that the most of

variance is due to few high peaks, as opposed to the undesirable situation of frequent small deviations from the mean time to fix; these high peaks are of course desired in the left side of the distribution, indicating that a lot of defects have a short time to fix: thus a good value of kurtosis does not suffice by itself, but the skewness matters too. In particular, a positive skew is desired, as in our case, indicating that the time to fix distribution is right-tailed, with a lot of defects with low time to fix. An example of distribution of the number of issues according to the TTFix is shown in Figure 4 for minor issues. It can be observed that the distribution is right-tailed with a pronounced peak in the left side. The distribution of major and blocking issues, here omitted, exhibit an even better shape, as expected by looking at their kurtosis and skew values.

Table 3: Statistics of the TTFix distributions. Mean values and standard deviations are measured in days.

Severity	$MTTFix$	MED_{TTFix}	$StdDev_{TTFix}$	MQT	MED_{QT}	$StdDev_{QT}$	$\#Res$	$Kurtosis$	$Skewness$
Minor	15.88	6.16	29.29	5.67	0.89	16.39	3.69	42.81	6.29
Major	12.69	5.76	25.28	3.42	0.81	9.98	3.75	52.26	6.85
Blocking	9.85	4.70	21.42	2.46	0.65	8.42	3.72	79.89	8.24

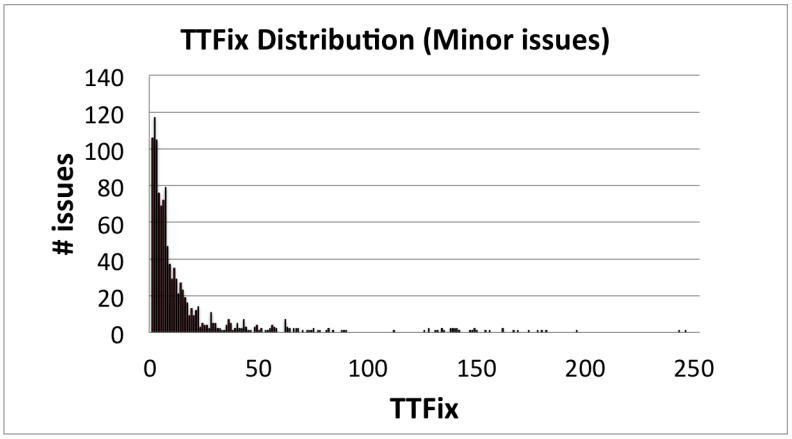


Fig. 4: Distribution of the number of minor issues according to the TTFix.

Regarding the number of resources allocated to defect, we can note that on average 3.7 resources are allocated on defects, irrespective from their severity. Finally, considering the queuing time, again we can observe that the MQT of blocking issues is shorter than the one of major and minor ones, as desirable.

However, this time is not negligible and equals to about one third of the overall debug process on average. Hence, estimations based solely on the measure of the time to process the issue, ignoring the queuing time, could end up with inaccurate results.

5.2 Impact of non-immediate debug

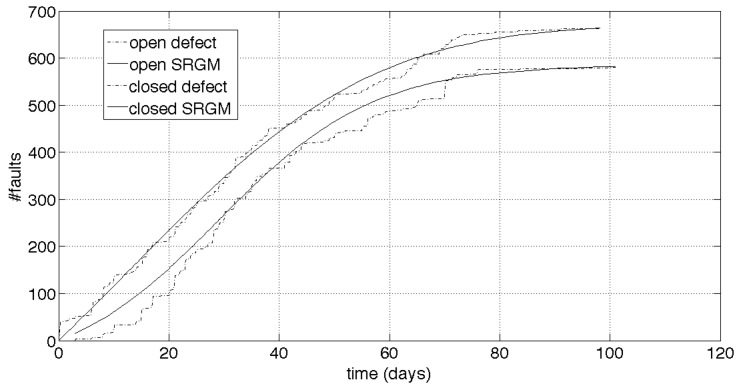
This Section discusses the impact of debugging on the SRGM-based analysis. Synthetic statistics about the observed debugging process tell that its overall features reflect a good process, with low average and median time to fix, a net shape of the distribution, severities times to fix as expected, and so on. Figure 5(a) and 5(b) report the raw data about the cumulative number of opened (testing process) and closed (debugging process) issues, along with SRGMs fitting them. The graphs show what is the impact of the debugging times on the achieved quality.

The closed defect curve is the one actually contributing to reliability increase (namely, when the defect is actually removed); the opening curve would represent the reliability increase only under immediate repairs. Thus, in the following, we consider the difference between the two curves and their corresponding models in order to infer conclusions about the debugging impact.

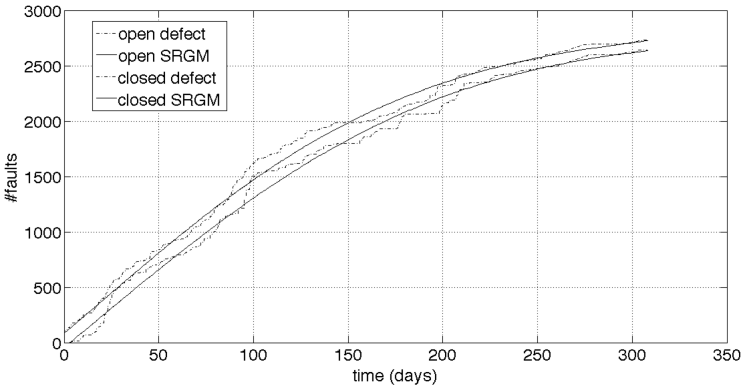
Let us define $\Delta_{issues}(t)$ and $\Delta_{time}(F)$, respectively, as: *i)* the difference between the opened and closed issues at a fixed time t (thus, *pending issues* at t, which is the vertical distance between the raw data curves), and *ii)* the time required to close a given number of opened issues, F (namely, the *delay* of the debug process compared to testing, which is the horizontal distance between the raw data curves). We also define the differences between the corresponding models as $\rho_{mvf}(t)$ and $\rho_{time}(F)$. These are the differences between the opened and closed issues at time t and at $mvf = F$, respectively, as estimated/predicted by the corresponding SRGMs. The Δ values are used to: *i)* evaluate the difference between the actually achieved quality (in terms of number of closed issues) and the believed one[4], which is the quality under immediate repair assumption (i.e., the opened issues), as well as *ii)* the difference between the actual time required to close F issues and the believed time (again, under immediate debugging, through the opening curve). This is the impact of assuming immediate debugging on quality/time estimates. On the other hand, the ρ values are used to assess the same differences on *predicted* values, which are needed to take decisions like "when to stop testing". This is the impact of the immediate debugging assumption on predictions made through SRGMs.

The version 1 has already detected 99% of the total estimated issues and has been released, while the version 2 is still at 94% and is still to release: thus,

[4] Quality in the following is expressed through the (predicted) number of closed issues or the (predicted) percentage of closed issues with respect to the total one; for what said previously about the equivalence of this information to failure intensity and thus reliability, "quality estimation" and "quality prediction" are equivalent to "reliability estimation" and "reliability prediction"



(a) Cumulative number of opened and closed issues for version 1



(b) Cumulative number of opened and closed issues for version 2

Fig. 5: Cumulative number of opened-closed issues and fitted SRGMs for both versions

we compute on version 1 the Δ differences on actual data to see the impact on estimated quality/time, whereas, on version 2, we compute the ρ values on future predictions, at percentages greater than the achieved 94%.

Let us first consider the version 1. At the last day, 98.19, the total opened issues were 665, namely about the 99% of total estimated ones. The actual quality at that time is given by the closed issues, that are 578 , thus the 86.14 % of the total estimated one, rather than the believed 99%. The error is therefore:

$$\epsilon_{\Delta_{issues}} = \frac{\Delta_{issues}(98.19)}{Closed(98.19)} \cdot 100 = \frac{665 - 578}{578} \cdot 100 = 15.05\% \quad (1)$$

where $Closed(t)$ is the number of closed issues at time t.
This means that if tester released actually at 99% of total issues and use the opening curve assuming immediate repair, the release quality is overestimated of 15.05 %. Similarly, if tester used the opening curve assuming immediate repair, the removed 578 issues occurred, in his view, after 63.98 days, rather than at 98.19; thus there is a time estimation error of:

$$\epsilon_{\Delta_{time}} = \frac{\Delta_{time}(578)}{ClosedTime(578)} \cdot 100 = \frac{98.19 - 63.98}{98.19} \cdot 100 = 34.84\% \quad (2)$$

where $ClosedTime(F)$ is the time of closing of the $F - th$ issus. This is interpreted as: using the immediate debugging assumption, the required quality is reached 34.84% later than the believed time. The first raw of Table 4 reports results for release quality values from 95% to 98%, besides the mentioned 99% case.

Table 4: Results on the impact of debugging time on both versions

Version	*Achieved or Predicted release quality*				
	95%	**96%**	**97%**	**98%**	**99%**
Version 1: $\epsilon_{\Delta_{issues}}$	14.15%	14.59%	13.61%	14.06%	15.05%
Version 1: $\epsilon_{\Delta_{time}}$	14.63%	16.07%	20.37%	31.28%	34.84%
Version 2: $\epsilon_{\rho_{mvf}}$	0.04%	0.18%	0.33%	0.63%	1.05%
Version 2: $\epsilon_{\rho_{time}}$	0.31%	1.18%	3.58%	14.89%	∞

On the other hand. If tester has not achieved a high quality level yet, s/he may want to use SRGMs for a prediction purpose and decide on when to stop testing. This is well represented by version 2. In this case, detected issues have been 2647, namely the 94.4 % of total estimated ones. We evaluate the impact of debugging time on prediction accuracy supposing that tester wants to release at 95%, 96%, 97%, 98%, and 99% of total defects (namely: 2662, 2690, 2718, 2746, 2774 defects). In these cases, if s/he uses the opening curve, the release should be at the days: 321, 339, 363, 396, and 447.

But using the closing curve, to those times it corresponds: 2661, 2685, 2709, 2729, 2745 of removed issues, which will cause quality overestimation errors. For instance, suppose that tester wants to release at 97%. In this case, the quality overestimation error will be:

$$\epsilon_{\rho_{mvf}} = \frac{\rho_{mvf}(363)}{SRGM(Closed(363))} \cdot 100 = \frac{2718 - 2709}{2709} \cdot 100 = 0.33\% \qquad (3)$$

Similarly to the version 1 case, there will also be an error about the time prediction. If tester uses the opening curve assuming immediate debugging to release at 97%, the opened 2718 issues in 363 days will be closed (looking at the closing curve) only at day 376, causing an error of[5]:

$$\epsilon_{\rho_{time}} = \frac{\rho_{time}(2718)}{SRGM(ClosedTime(2718)} \cdot 100 = (376 - 363)/363 \cdot 100 = 3.58\% \quad (4)$$

where $SRGM(ClosedTime(F))$ is the predicted time required to close F issues. The second part of Table 4 reports results form 95% to 99% release criteria. As may be noticed, the errors on quality overestimation are quite small in version 2, compared to version 1, and are slightly increasing with the desired quality. The small error denotes a very good debugging process, whose curve is strictly following the opening one. Notwithstanding, it is interesting to note how the error on the time prediction is higher, and increases rapidly for increasing values of the desired release quality, due to the saturation of both curves. From 98% to 99%, it increases up to infinite. This is interpreted as follows: if tester wants to release at 98% of the total estimated issues, and uses the opening curve assuming immediate repair, it would predict a testing time of 14.89% days less than the actually required testing time. If this desired quality goes beyond the 98%, such an error increases abruptly, reaching infinite at 99%. Thus, depending on the desired quality and on debugging process characteristics, this *testing time underestimation error* may be very high and is much more sensitive than the *quality overestimation error*.

In general, such time error always goes to infinite at some point (precisely, at the saturation point of the closing curve); in the practice, it can go to infinite considerably earlier if the debug process is not as close to the testing process as in the version 2 case. For instance, in version 1, for the same type of error (computed on raw data) the infinite occur soon after 578 issues, i.e., at only 86.14 % of the total estimated ones.

To summarize, the worse the debugging process, the greater the error on quality estimation is, and the earlier the time prediction error goes to infinite: but

[5] Note that, unlike the case of Δ_{time} value, here the difference is taken between the predicted time to close the number of issues that tester wants to remove and the predicted time to open that number of issues. For Δ_{time} values, we take the time to close the number of issues actually closed subtracted by the time at which that number of issues was opened (i.e., the "believed" time for achieving that quality).

while the *quality estimation/prediction error* is directly related to the number of pending issues quite independently from the release time (e.g., in the same way at 70%, 80%, or 90%), the *time estimation/prediction error* is much more sensitive: at high quality values, the underestimation of the required testing time can be very high, depending on the saturation of the opening and closing curves.

6 Conclusion

In this paper we analyzed the impact of the debugging time on reliability estimation and prediction. Characterization of issues found in a real-world industrial project indicates that the time taken by the debugging process is not negligible. For example, the debugging time causes an overestimation of the perceived software quality up to 15.5% in our dataset; similarly, it causes the underestimation of the testing time that would be required to obtain a given quality for a software product. In the future we aim to analyze further dataset and issues found in industrial projects in order to achieve a representative characterization of the debugging time and more accurate reliability estimation.

Acknowledgment

This work has been partially supported by the European Commission in the context of the FP7 project "ICEBERG", Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 324356.

References

1. C. Stringfellow, A. Amschler Andrews: An Empirical Method for Selecting Software Reliability Growth Models. Empirical Software Engineering, 7 (4) (2002)
2. W. Farr: Handbook of Software Reliability Engineering, M.R. Lyu (Ed.), chapter: Software Reliability Modeling Survey, pp. 71–117. McGraw-Hill, New York, NY (1996)
3. J.D. Musa, K. Okumoto: A logarithmic Poisson execution time model for software reliability measurement. In Proc. 7th Int. Conf. on Software Engineering (ICSE), pp. 230–238 (1984)
4. B. Zachariah, R.N. Rattihalli: Failure Size Proportional Models and an Analysis of Failure Detection Abilities of Software Testing Strategies. IEEE Trans. on Reliability, vol.56, no.2 (2007)
5. J. B. Dugan, Automated Analysis of Phase-Mission Reliability, IEEE Transaction on Reliability, vol. 40, 45-52, 1991.
6. Garzia, M.R., Assessing the Reliability of Windows Servers, Proc. of IEEE Dependable Systems and Networks, (DSN-2002).
7. Pietrantuono, R.; Russo, S.; Trivedi, K.S., "Online Monitoring of Software System Reliability," Dependable Computing Conference (EDCC), 2010 European , vol., no., pp.209,218, 28-30 April 2010, doi: 10.1109/EDCC.2010.33

8. Cotroneo, D.; Pietrantuono, R.; Russo, S., "Combining Operational and Debug Testing for Improving Reliability," Reliability, IEEE Transactions on , vol.62, no.2, pp.408,423, June 2013, doi: 10.1109/TR.2013.2257051
9. A.L. Goel, K. Okumoto: Time-dependent error-detection rate model for software reliability and other performance measures. IEEE Trans. on Reliability, R-28(3) (1979)
10. S. Yamada, M. Ohba, S. Osaki: S-Shaped Reliability Growth Modeling for Software Error Detection. IEEE Trans. on Reliability, R-32(5) (1983)
11. S.S. Gokhale, K.S. Trivedi: Log-logistic software reliability growth model. In: Proc. 3rd Int. High-Assurance Systems Engineering Symposium, pp. 34–41 (1998)
12. S. Yamada, H. Ohtera, and H. Narihisa: Software reliability growth models with testing effort. IEEE Trans. on Reliability, vol. R-35 (1986)
13. A. L. Goel. Software Reliability Models: Assumptions, Limitations and Applicability. IEEE Trans. on Software Engineering, SE-11(12) (1985)
14. H. Okamura, Y. Watanabe, T. Dohi, 2003. An iterative scheme for maximum likelihood estimation in software reliability modeling. In: Proc. 14th Int. Symp. on Software Reliab. Eng. (ISSRE-2003). IEEE CS Press, pp. 246256.
15. K. Ohishi, H. Okamura, T. Dohi: Gompertz software reliability model: Estimation algorithm and empirical validation. Journal of Systems and Software, 82 (3) (2009)
16. Madhu Jain, T. Manjula, T. R. Gulati, Software Reliability Growth Model (SRGM) with Imperfect Debugging, Fault Reduction Factor and Multiple Change-Point, Proceedings of the International Conference on Soft Computing for Problem Solving (SocProS 2011), 2011, Deep, Kusum and Nagar, Atulya and Pant, Millie and Bansal, Jagdish Chand Eds. Advances in Intelligent and Soft Computing, Springer, pp. 1027-1037.
17. J. D. Musa, A. Iannino, and K. Okumoto, Software Reliability, Measurement, Prediction and Application, McGraw Hill, 1987.
18. Chin-Yu Huang, Wei-Chih Huang, Software Reliability Analysis and Measurement Using Finite and Infinite Server Queueing Models, Reliability, IEEE Transactions on (Volume:57, Issue: 1), 2008. 192–203.
19. Tung Thanh Nguyen and T.N. Nguyen and E. Duesterwald and T. Klinger and P. Santhanam, Software Engineering (ICSE), 2012 34th International Conference on, 2012. 1297 –1300
20. Feng Zhang and F. Khomh and Ying Zou and A.E. Hassan, An empirical study on factors impacting bug fixing time, Reverse Engineering (WCRE), 19th Working Conference on, 2012.
21. Akinori Ihara and Masao Ohira and Kenichi Matsumoto, An analysis method for improving a bug modification process in open source software development, Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops, ACM. 2009, 135–144.
22. Pietrantuono, R.; Russo, S.; Trivedi, K.S., "Software Reliability and Testing Time Allocation: An Architecture-Based Approach," Software Engineering, IEEE Transactions on , vol.36, no.3, pp.323,337, May-June 2010, doi: 10.1109/TSE.2010.6
23. M. Cinque, D. Cotroneo, and A. Pecchia, "Event Logs for the Analysis of Software Failures: A Rule-Based Approach". Software Engineering, IEEE Transactions on, vol.39, no.6, pp.806-821, June 2013
24. F. Frattini, R. Ghosh, M. Cinque, A. Rindos, and K. S. Trivedi, "Analysis of bugs in Apache Virtual Computing Lab," 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp.1-6, 2013