Increment Software Reliability Using Bug Cycle And Duplicate Detection

Miss. Poorva Sabnis[1], Mr. Amol Kadam[2], Dr. S. D. Joshi[3]

[1] Bharati Vidyapeeth Deemed University College of Engineering, Pune

[2] Bharati Vidyapeeth Deemed University College of Engineering, Pune

[3]Bharati Vidyapeeth Deemed University College of Engineering, Pune

Abstract

Software Reliability is defined as the probability of free-failure operation for a specified period of time in a specified environment in a given period of time under specified conditions. Software Reliability Growth models (SRGM) have been developed to estimate software reliability measures such as number of remaining faults, software failure rate and software reliability. Software testing can be defined as a process to detect faults in the totality and worth of developed computer software. Testing is very important in assuring the quality of the software by identifying faults in software, and possibly removing them. In this paper, we are focusing on increasing the reliability of the software using bug tracking system. In this bug tracking system, we are including 2 methods as – bug cycle for bug detection and bug duplication avoiding technique. In bug cycle, we are going to investigate that, when verification is performed, who performs the verification and how verification performed. In duplicate detection, we propose a system that automatically classifies duplicate bug reports as they arrive to save developer time. Our system is able to reduce development cost by filtering out 8% of duplicate bug reports.

Keywords: SDLC, SRGM, bug cycle, duplicate detection

1. Introduction

Software Development Lifecycle Models

A software development lifecycle is a structure imposed on the development of a software product. Synonyms include development lifecycle and software process. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

There are various models present in software development as waterfall model, iterative model, spiral model, RAD(Rapid Application Development)etc. Generally these models contains various stages of development as – Requirement analysis and development, System and software design, Coding, Testing, Quality Management, Maintenance etc. All these phases are very important to develop any software. We are here focusing on the most important phase i.e. testing phase. The aim is to develop a software in such a way that is should contain less number of errors. Hence we are trying to minimize the errors in the software in testing phase itself by using software reliability growth model.

Software Reliability Growth Models

Software reliability is a field of testing which deals with checking the ability of software to function under given environmental conditions for a particular amount of time, taking into account the precision of the software. In software reliability testing, problems are discovered regarding software design and functionality and assurance is given that the system meets all requirements. Software reliability is the probability that software will work properly in a specified environment and for a given time. Using the following formula, the probability of failure is calculated by testing a sample of all available input states.

Probability = Number of failing cases / Total number of cases under consideration

Importance of reliability testing:

The application of computer software has crossed into many different fields, with software being an essential part of industrial, commercial and military systems. Because of its many applications in safety critical systems, software reliability is now an important research area. Although software engineering is becoming the fastest developing technology of the last century, there is no complete, scientific, quantitative measure to assess them. Software reliability testing is being used as a tool to help assess these software engineering technologies.

To improve the performance of software product and software development process, a thorough assessment of reliability is required. Testing software reliability is important as it is of great use for software managers and practitioners. We are going to use 2 methods for reliability as- bug cycle and duplicate detection.

1.1 Bug Cycle

Bug repositories have for a long time been used in software projects to support coordination among stakeholders. They record discussion and progress of software evolution activities, such as bug fixing and software verification. Hence, bug repositories are an opportunity for researchers who intend to investigate issues related to the quality of both the product and the process of a software development team. However, mining bug repositories has its own risks.
Previous research has identified problems of missing data (e.g., rationale, traceability links between reported bug fixes and source code changes) [1], inaccurate data (e.g., misclassification of bugs) [2], and biased data [3]. In previous research, we tried to assess the impact of independent verification of bug fixes on software quality, by mining data from bug repositories. We relied on reported verifications tasks, as recorded in bug reports, and interpreted the recorded data according to the documentation for the specific bug tracking system used. Hence, in this paper, we investigate the following exploratory research questions regarding the software verification process:

• **When is the verification performed**: is it performed just after the fix, or is there a verification phase?

• **Who performs the verification**: is there a QA (quality assurance) team?

• **How is the verification performed**: are there performed ad hoc tests, automated tests, code inspection?

Bug tracking systems allow users and developers of a software project to manage a list of bugs for the project, along with information such as steps to reproduce the bug and the operating system used. Developers choose bugs to fix and report on the progress of the bug fixing activities, ask for clarification, discuss causes for the bug etc. One important feature of a bug that is recorded on bug tracking systems is its status. The status records the progress of the bug fixing activity. Figure 1 shows each status that can be recorded, along with typical transitions between status values, i.e., the workflow.

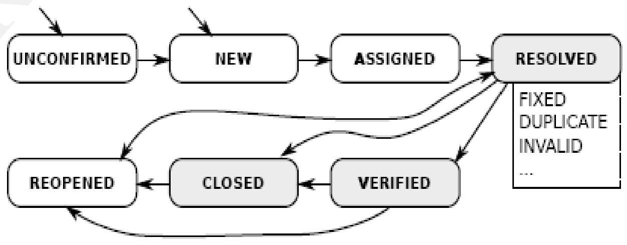

Figure 1. Simplified workflow for Bugzilla. Adapted from http://www.bugzilla.org/docs/2.18/html/lifecycle.html.

In simple cases, a bug is created and receive the status UNCONFIRMED (when created by a regular user) or NEW (when created by a developer). Next, it is ASSIGNED to a developer, and then it is RESOLVED, possibly by fixing it with a patch on the source code. The solution is then

VERIFIED by someone in the quality assurance team, if it is adequate, or otherwise it is REOPENED. When a version of the software is released, all VERIFIED bugs are CLOSED.

It states that, any bug is 'VERIFIED' means that QA [quality assurance team] has looked at the bug and the resolution and agrees that the appropriate resolution has been taken". It does not specify how developers should look at the resolution (e.g., by looking at the code, or by running the patched software).

1.2) Duplicate Detection

Also we are going to include one more facility as "Duplicate detection" for bug tracking system.

Bug tracking systems are important tools that guide the maintenance activities of software developers. The utility of these systems is hampered by an excessive number of duplicate bug reports–in some projects as many as a quarter of all reports are duplicates. Developers must manually identify duplicate bug reports, but this identification process is time-consuming and exacerbates the already high cost of software maintenance. We propose a system that automatically classifies duplicate bug reports as they arrive to save developer time. Our system is able to reduce development cost by filtering out 8% of duplicate bug reports while allowing at least one report for each real defect to reach developers. We propose a technique to reduce bug report triage cost by detecting duplicate bug reports as they are reported. We build a classifier for incoming bug reports that combines the surface features of the report [6], textual similarity metrics [15], and graph clustering algorithms [10] to identify duplicates. We attempt to predict whether manual triage efforts would eventually resolve the defect report as a duplicate or not. This prediction can serve as a filter between developers and arriving defect reports: a report predicted to be a duplicate is filed, for future reference, with the bug reports it is likely to be a duplicate of, but is not otherwise presented to developers. As a result, no direct triage effort is spent on it. Our classifier is based on a model that takes into account easily-gathered surface features of a report as well as historical context information about previous reports.

2) Motivating example

Bug tracking systems allow users and developers of a software project to manage a list of bugs for the project, along with information such as steps to reproduce the bug and the operating system used. Developers choose bugs to fix and report on the progress of the bug fixing activities, ask for clarification, discuss causes for the bug etc. In this research, we focus on Bugzilla, an open source bug tracking system used by software projects such as Eclipse, Mozilla, Linux Kernel, NetBeans, Apache, and companies such as NASA and Facebook. The general concepts from Bugzilla should apply to most other bug tracking systems.

Software verification techniques are classified in static and dynamic [5]. Static techniques include source code inspection, automated static analysis, and formal verification. Dynamic techniques, or testing, involve executing the software system under certain conditions and comparing its actual behavior with the intended behavior. Testing can be done in an improvised way (ad hoc testing), or it can be structured as a list of test cases, leading to automated testing.

Duplicate bug reports are such a problem in practice that many projects have special guidelines and websites devoted to them. The "Most Frequently Reported Bugs" page of the Mozilla Project's Bugzilla bug tracking system is one such example. This webpage tracks the number of bug reports with known duplicates and displays the most commonly reported bugs. Ten bug equivalence classes have over 100 known duplicates and over 900 other equivalence classes

have more than 10 known duplicates each. All of these duplicates had to be identified by hand and represent time developers spent administering the bug report database and performing triage rather than actually addressing defects.

Bug report #340535 is indicative of the problems involved; we will consider it and three of its duplicates. The body of bug report #340535, submitted on June 6, 2006, includes the text, "when I click OK the updater starts again and tries to do the same thing again and again. It never stops. So I have to kill the task." It was reported with severity "normal" on Windows XP and included a log file.

Bug report #344134 was submitted on July 10, 2006 and includes the description, "I got a software update of Minefield, but it failed and I got in an endless loop." It was also reported with severity "normal" on Windows XP, but included no screenshots or log files. On August 29, 2006the report was identified as a duplicate of #340535.

3) Method

3.1) Bug Cycle : In order to answer the research questions—when and how bug fixes are verified, and who verifies them—, a three-part method was used:

1) **Data extraction**: we have obtained publicly available raw data from the Bugzilla repositories.

2) **Data sampling**: for each project, two representative subprojects were chosen for analysis.

3) **Data analysis**: for each research question, a distinct analysis was required, as will be further described.

A. Data Extraction

In order to perform the desired analyses, we needed access to the data recorded by Bugzilla for a specific project, including status changes and comments. Bugzilla is a particularly popular open

source bug tracking software system. Bugzilla bug reports come with a number of pre-defined fields, including categorical information such as the relevant product, version, operating system and self-reported incident severity, as well as free-form text fields such as defect title and description. In addition, users and developers can leave comments and submit attachments, such as patches or screenshots.

B.Data Sampling

The Platform subprojects are the main subprojects for the respective IDEs, so they are both important and representative of each projects' philosophy. The other two subprojects were chosen at random, restricted to subprojects in which the proportion of verified bugs was greater than the proportion observed in the respective Platform subprojects. The reason is to avoid selecting projects in which bugs are seldom marked as VERIFIED.

C. Analysis: When Are Bugs Verified?

In order to determine if there is a well-defined verification phase for the subprojects, we have selected all reported verifications (i.e., status changes to VERIFIED) over the lifetime of each subproject. Then, we have plotted, for each day in the interval, the accumulated number of verifications reported since the first day available in the data. The curve is monotonically increasing, with steeper ascents representing periods of intense verification activity. Also, we have obtained the release dates for multiple versions of Eclipse and NetBeans. The information was obtained from the respective websites. In cases in which older information was not available, archived versions of the web pages were accessed via the website www.archive.org.

If a subproject presents a well-defined verification phase, it is expected that the verification activity is more intense a few days before a release. Such pattern can be identified by visual inspection of the graph, by looking for steeper ascents in the verification curve preceding the release dates.

D. Analysis: Who Verifies Bugs?

In order to determine whether there is a team dedicated to quality assurance (QA), we have counted how many times each developer has marked a bug as FIXED or VERIFIED. We considered that a developer is part of a QA team if s/he verified at least 10 times (i.e., one order of magnitude) more than s/he fixed bugs. Also, we have computed the proportion of verifications that was performed by the discovered QA team. It is expected that, if the discovered set of developers is actually a QA team, they should be responsible for the majority of the verifications.

E. Analysis: How Are Bugs Verified?

In order to discover the verification techniques used by the subprojects, we have selected the comments written by developers when they mark a bug as VERIFIED (meaning that the fix was accepted) or REOPENED (meaning that the fix was rejected). The comments were matched against five regular expressions, each corresponding to one of the following verification techniques: automated testing, source code inspection, ad hoc testing, automated static analysis, and formal testing.

The complete regular expressions are available in the experimental package. It should be noted that regular expressions may not be a reliable alternative to the problem of identifying verification techniques in comments. In future research, more advanced information retrieval techniques should be explored. Nevertheless, regular expressions enable an initial study and help unveil insights about verification techniques in bug reports.

3.2) Duplicate detection

Our goal is to develop a model of bug report similarity that uses easy-to-gather surface features and textual semantics to predict if a newly-submitted report is likely to be a duplicate of a previous report. Since many defect reports are duplicates (e.g., 25.9% in our dataset), automating this part of the bug triage process would free up time for developers to focus on other tasks, such as addressing defects and improving software dependability.

Our formal model is the backbone of our bug report filtering system. We extract certain features from each bug report in a bug tracker. When a new bug report arrives, our model uses the values of those features to predict the eventual duplicate status of that new report. Duplicate bugs are not directly presented to developers to save triage costs. We employ a linear regression over properties of bug reports as the basis for our classifier. Linear regression offers the advantages of (1) having off-the-shelf software support, decreasing the barrier to entry for using our system; (2)

supporting rapid classifications, allowing us to add textual semantic information and still perform real-time identification; and (3) easy component examination, allowing for a qualitative analysis of the features in the model. Linear regression produces continuous output values as a function of continuously-valued features; to make a binary classifier we need to specify those features and an output value cutoff that distinguishes between duplicate and non-duplicate status.

1) Textual Analysis

Bug reports include free-form textual descriptions and titles, and most duplicate bug reports share many of the same words. Our first step is to define a textual distance metric for use on titles and descriptions. We use this metric as a key component in our identification of duplicates.

We adopt a “bag of words” approach when defining similarity between textual data. Each text is treated as a set of words and their frequency: positional information is not retained.

Since orderings are not preserved, some potentially important semantic information is not available for later use. The benefit gained is that the size of the representation grows at most linearly with the size of the description. This reduces processing load and is thus desirable for a real-time system.

We treat bug report titles and bug report descriptions as separate corpora. We hypothesize that the title and description have different levels of importance when used to classify duplicates. In our experience, bug report titles are written more succinctly than general descriptions and thus

are more likely to be similar for duplicate bug reports. We would therefore lose some information if we combined titles and descriptions together and treated them as one corpus.

We pre-process raw textual data before analyzing it, tokenizing the text into words and removing stems from those words. We use basic scripting to obtain tokenized, stemmed word lists of description and title text from raw defect reports. Tokenization strips punctuation, capitalization, numbers, and other non-alphabetic constructs. Stemming removes inflections(e.g.,“scrolls” and “scrolling” both reduce to “scroll”).

Stemming allows for a more precise comparison between bug reports by creating a more normalized corpus; our experiments used the common Porter stemming algorithm. We then filter each sequence against a stoplist of common words. Stoplists remove words such as “a” and “and” that are present in text but contribute little to its comparative meaning. If such words were allowed to remain, they would artificially inflate the perceived similarity of defect reports with long descriptions. Finally, we do not consider submission-related information, such as the version of the browser used by the reporter to submit the defect report via a web form, to be part of the description text. Such information is typically collocated with the description in bug databases, but we include only textual information explicitly entered by the reporter. In this we are going to use 3 methods as following: 1) Document Similarity 2) Weighting for duplicate defect detection 3) clustering.

2) Model Features

We use textual similarity and the results of clustering as features for a linear model. We keep description similarity and title similarity separate. For the incoming bug report under consideration, we determine both the highest title similarity and highest description similarity it shares with a report in our historical data. Intuitively, if both of those values are low then the incoming bug report is not textually similar to any known bug report and is therefore unlikely to

be a duplicate. We also use the clusters from Section 4.1.3 to define a feature that notes whether or not a report was included in a cluster. Intuitively, a report left alone as a singleton by the clustering algorithm is less likely to be a duplicate. It is common for a given bug to have multiple duplicates, and we hope to tease out this structure using the graph clustering. Finally, we complete our model with easily-obtained surface features from the bug report. These features include the self-reported severity, the relevant operating system, and the number of associated patches or screenshots. These features are neither as semantically-rich nor as predictive as textual similarity. Categorical features, such as relevant operating system, were modeled using a one-hot encoding.

So finally we conclude four empirical evaluations:

Text. Our first experiment demonstrates the lack of correlation between sharing “rare” words and duplicate status. In our dataset, two bug reports describing the same bug were no more likely to share "rare" words than were two non-duplicate bug reports. This finding motivates the form

of the textual similarity metric used by our algorithm.

Recall. In this experiment, each algorithm is presented with a known-duplicate bug report and a set of historical bug reports and is asked to generate a list of candidate originals for the duplicate. If the actual original is on the list, the algorithm succeeds. We perform no worse than the current state of the art.

Filtering. Our third and primary experiment involved on-line duplicate detection. We tested the feasibility and effectiveness of using our duplicate classifier as an on-line filter. We trained our algorithm on the first half of the defect reports and tested it on the second half. Testing proceeded chronologically through the held-out bug reports and predicted their duplicate status. We measured both the time to process an incoming defect report as well as the expected

savings and cost of such a filter. We measured cost and benefit in terms of the number of real defects mistakenly filtered as well as the number of duplicates correctly filtered.

Features. Finally, we applied a leave-one-out analysis and a principal component analysis to the features used by our model. These analyses address the relative predictive power and potential overlap of the features we selected.

4) Conclusions

So we have found, using only data from bug repositories, subprojects with and without QA teams, with and without a well-defined verification phase. We also have found weaker evidence of the application of automated testing and source code inspection. Also, there were cases in which marking a bug as VERIFIED did not imply that any kind of software verification was actually performed. We propose a system that automatically classifies duplicate bug reports as they arrive to save developer time. This system uses surface features, textual semantics, and graph clustering to predict duplicate status. We empirically evaluated our approach using a dataset of 29,000 bug reports from the Mozilla project, a larger dataset than has generally previously been reported. We show that inverse document frequency is not useful in this task, and we simulate using our model as a filter in a real-time bug reporting environment. Our system is able to reduce development cost by filtering out 8% of duplicate bug reports. It still allows at least one report for each real defect to reach developers, and spends only 20 seconds per incoming bug report to make a classification.

5) References

[1] Characterizing Verification of Bug Fixes in Two Open Source IDEs

Rodrigo Souza and Christina Chavez Software Engineering Labs

Department of Computer Science – IM, Universidade Federal da Bahia (UFBA), Brazil, {rodrigo,flach}@dcc.ufba.br 978-1-4673-1761-0/12/$31.00 c 2012 IEEE

[2] Automated Duplicate Detection for Bug Tracking Systems

Nicholas Jalbert. University of Virginia, Charlottesville, Virginia 22904, jalbert@virginia.edu
Westley Weimer University of Virginia, Charlottesville, Virginia 22904, weimer@cs.virginia.edu

International Conference on Dependable Systems & Networks: Anchorage, Alaska, June 24-27 2008

[3] J. Aranda and G. Venolia, "**The secret life of bugs: Going past the errors and omissions in software repositories,**" in Proc. Of the 31st Int. Conf. on Soft. Engineering, 2009, pp. 298–308.

[4] J. Anvik, L. Hiew, and G. C. Murphy. **Who should fix this bug**? In International Conference on Software Engineering (ICSE), pages 361–370, 2006.

[5] I. Sommerville, Software engineering (5th ed.). Addison Wesley Longman Publish. Co., Inc., 1995.