Evaluating a Tool for Bug-report Analysis and Search

Yguaratã Cerqueira Cavalcanti\textsuperscript{1,3}, Paulo Anselmo da Silveira Mota\textsuperscript{1,3}, Ivan do Carmo Machado\textsuperscript{2,4}, Eduardo Santana de Almeida\textsuperscript{2,3}
Silvio Romero de Lemos Meira\textsuperscript{1,3}

\textsuperscript{1}Center for Informatics – Federal University of Pernambuco (UFPE)
\textsuperscript{2}Computer Science Department – Federal University of Bahia (UFBA)
\textsuperscript{3}Reuse in Software Engineering (RiSE)

\{ycc, pamsn, srlm\}@cin.ufpe.br, \{ivan, esa\}@dcc.ufba.br

Abstract. Bug report tracking systems have been used to facilitate the maintenance and evolution of software. However, duplicate entries of bug reports in such systems can considerably impact productivity within software project. This reduction in productivity occurs because duplicate entries demand more time for search and analysis of bug reports. In this context, this paper presents the main problems caused by bug report duplication problem. In addition, a tool for bug reports search and analysis (BAST) is proposed to deal with the duplication avoidance, as well as, it also presents a case study to evaluate the tool. For the evaluation, we compared BAST against a baseline tool in a private company for software testing. The results showed that BAST worked better than the other one, both to reduce the time of analysis, as well as, to reduce the number of duplicates submitted.

1. Introduction

To improve software maintenance and evolution organizations use specific systems to manage, track and store change requests within software projects. These systems are generally called bug report tracking systems, bug repositories, or just bug trackers [13, 18]. A bug report is defined as a software artifact that describes some defect or enhancement that is submitted to a bug tracker by developers, users, testers, and other possible stakeholders. Such systems are beneficial as they allow software changes to be identified and reported quickly [2]. Furthermore the bug report historical databases can be a useful source of documentation for the project.

Typically, each bug report contains information, such as a bug identifier, summary and detailed description, software version and component in which the bug appeared, described dependencies with other bug reports (i.e. duplicate bug reports) and the assigned developer. During the life-time of a bug report, continued updates on the bug status, description and additional comments can be added to help resolve the bug.

Although bug trackers bring many benefits to software development, some challenges may appear as a consequence of its usage, such as: the dynamic assignment of bug reports [1], the quality of bug report descriptions [5], and the detection of duplicate bug reports [19]. Bug trackers also present opportunities to improve software development, such as using the bug tracker historical database to: analysis of the impact of change and effort estimation [20] and software evolution and traceability [14].
This work is focused on the problem of *bug report duplication problem*. This problem is characterized by the submission of two or more bug reports that describe the same change request. The main consequence of bug report duplication problem is the time-cost associated with managing these redundant bug reports. For example, the staff responsible for the search and analysis of bug reports spend a considerable amount of time dealing with bug reports that have already been submitted. Recent work [2, 7, 16] has shown that between 10% to 30% of a bug report repository is composed of duplicate bug reports. For more information about the factors and consequences of duplicate bug reports in software projects, we recommend to read our previous researches in [7, 8].

This paper presents a case study performed in order to evaluate a tool (BAST – Bug Report Analysis and Search Tool) for search and analysis of bug reports. BAST was built with the objective to reduce the amount of time needed to perform these tasks, and to reduce the number of duplicate bug reports that are frequently submitted. Although the related work presented different approaches for solving bug report duplication, very few experimentation against existing tools, or inside real environments, has been done to test them.

Moreover, the case studies present in related work does not considered about the time spent with these tasks. As we mentioned previously, one of the consequences of the presence of duplicate is the time spent during search and analysis, which we consider to be the second dimension of the duplication problem. Thus, this case study is of great contribution to the area.

The case study compared BAST against a baseline tool used inside a Motorola test center, located in Recife, Brazil. Both the amount of duplicates and time spent with search and analysis were analyzed in the case study. The results showed that BAST had better performance than the other one, thus reducing the time for search and analysis and the amount of duplicates submitted.

The remaining of this paper is organized as follow: Section 2 presents the proposed tool; Section 3 presents the case study performed inside Motorola test center; Section 4 presents the related work; and, finally, Section 5 summarizes the paper and presents some intended future work.

2. BAST: Bug Report Analysis and Search Tool

Bug Report Analysis and Search Tool (BAST) was developed to assist in the prevention of duplicate bug reports, as well as, to reduce the time to perform search and analysis of bug reports. Figure 1 shows a screenshot of BAST. The top of the figure contains a search field. The middle display shows a list of search results containing bug reports. The bottom of the figure displays details of the selected result from the middle screen.

In the search field, the user can specify some filters such as component name, status of the bug report, author identification (email or name). Moreover, the head of the search results list also allows the user to sort the results according to the various columns showed.

In addition to the search, filtering and visualization features, BAST implements Text Mining [10] techniques to prepare the bug reports prior to analysis and extraction of relevant information. Among such techniques, we implemented *stop words* removal.
To carry out searches in BAST, the user must specify keywords to characterize the information he/she seeks. The keyword-based search was chosen as it is a natural way to perform search and it is also a very intuitive way of expressing information needs [3].

BAST also extracts relevant information from the bug report’s content using natural language techniques [10]. For example, the tool can extract related bug reports to help identify duplicates, references to external links and names of other people involved informally with the bug report. These information are retrieved from the comments that people enter during the bug report life-time.

Finally, BAST was built to be easy adopted in different contexts. There are two ways of adopting it in a software development project: (a) to integrate the tool directly into the database of a bug tracker, or (b) to feed BAST with the files containing the bug reports exported from a bug tracker. Currently, BAST can import bug reports from major bug tracking tools such as Bugzilla, Mantis, Trac. Furthermore, BAST worked with company-specific bug tracking tools such as those used by Motorola. For more information about the tool, we recommend to read the work presented in [6].

3. Evaluating BAST: Case Study at Motorola

We performed a case study to evaluate BAST in a test center of the Motorola located in Recife, Brazil. In this test center, the testers perform a systematic process to test software that is developed by Motorola in various sites around the world. As the software development is distributed, tests are also performed distributed in different test centers. Briefly,
the testers receive a formal document with the tests that must be performed and, when errors are found, bug reports should be submitted. Next we describe the objectives of the case study, the baseline tool which BAST was compared with, the research questions of the study, the case study design, participants, and metrics. Then we provide the interpretation of the results, and discuss about the confound factors and lessons learned from the case study.

Case study objectives. The objective of this case study was to analyze the efficiency of BAST for search and analysis of bug reports, with a focus on avoiding duplicates, against a baseline tool. Thus, we want to: (i) examine whether BAST can prevent more duplicate bug reports than the baseline tool, and (ii) consider whether BAST decreases the time spent with search and analysis of bug reports.

Baseline tool. Currently, Motorola uses an internal tool to manage bug reports, as well as to search the existing ones. To performs searches in this tool, the submitters need to manually create multiple filters to specify the desired criteria for searches. Some of the negative points were raised by submitters on the current tool: the time and complexity to build the search filters, and the bug reports visualization that did not facilitate the identification of duplicates.

Research questions. We established the following research questions that we would like to response with this case study: a) Is BAST more effective than baseline tool to avoid duplicates? b) Can BAST reduce the time spent to search for similar bug reports?

Case study design. The case study had two treatments, and a specific participant (the submitter) was responsible to execute both of them. The submitter was instructed on how he should use the tools to search and analyze the bug reports. Thus, we divided the assessment period in two stages: the first stage (21 days), the submitter should carry out the search and analysis first using the baseline tool, and if he did not find a similar bug report, he should use BAST to perform the search and analysis; in the second stage (11 days), this sequence was reversed.

We designed the study in this manner to reduce the factor of confusion regarding the knowledge of already analyzed bug reports. For example, if we chose by conducting the study so that the submitter should perform search and analysis for the same bug reports in both tools, the analysis in the second tool would be compromised due to the submitter knowledge about possible existing similar bug reports.

Submitter selection. In order to carry out the case study, a specific tester, called bug report master, was selected to be the submitter who should use both tools in parallel during the whole period. The bug report master is the person responsible for conducting the test cycles. When a tester has problems or questions with regard to testing or bug report, the bug report master should be contacted to resolve the problems. Thus, this person, besides being responsible for the search and analysis of its own bug reports, is also responsible for doing that for people who contacted him. For example, if a tester has doubts with respect the uniqueness of a bug report, but cannot find a similar bug report in the repository, the bug report master should be contacted to make advanced search and analysis. For simplicity, from now on we will call the bug report master just as submitter.

Evaluation Metrics. To analyze which tool was more efficient regarding to the objectives defined, the following metrics were established:
Type of bug reports analyzed. With this metric we want to characterize the types of bug reports that were submitted during the course of the case study.

Number of duplicate bug reports avoided. By that metric we want to know how many bug reports that would be duplicated have been prevented with the use of our tool, and how many have been avoided with the use of the baseline tool.

Time spent to analyze similar bug reports. This metric is concerned with the time, in minutes, spent by submitters to search and analyze similar bug reports. For this metric, we computed only the time for searches that found similar bug reports.

In order to gather these metrics, the submitter was responsible for recording the types of bug reports that were analyzed, the time spent on each search and analysis, and specifying in which tool a similar bug report was found. If a bug report was not submitted and there were no similar bug reports for it, the submitter should specify the reason for not submitting. Next we describe and interpret the results of the case study.

3.1. Result Analysis

During the case study, it was examined 144 bug reports by the bug report master. This amount lead to 407 searches also performed by the bug report master, just in our tool during the study period. However, it was not possible to have access to the number of searches carried out in the Motorola’s tool due to issues of confidentiality. Next we describe the results considering the first and second stages of the case study execution, then we analyze the whole period.

3.1.1. Analysis of the First Stage

During the first stage, it was analyzed 42 bug reports. It was the stage with less bug reports to perform search and analysis. The main reason for that was due to the fact that it was the beginning of the project. Figure 2 shows a graph with the classification of these bug reports after the analysis performed by the submitter.

![Figure 2. Repository status in first stage.](image)

Number of duplicate bug reports avoided. Figure 3(a) shows the percentage of duplicate bug reports that were avoided according to the tool used. In this stage, the bug reports were analyzed using the baseline tool, and if it was not found similar bug reports, a new analysis should be performed using BAST. Thus, the analysis made with the baseline prevented the submission of 58% of duplicates, while BAST prevented 35%. In other words, the baseline tool was not able to avoid 35% of the total duplicates, which were
avoided using BAST. In addition, 7% were avoided because the information provided by developers. Therefore, during the first stage, the BAST had lower performance than the baseline tool.

**Average time spent on analysis of bug reports.** The graph in Figure 3(b) shows the average time spent with search and analysis in each tool. As it can be seen, the time to do analysis was considerably reduced with the use of BAST. The average time for search and analysis with BAST was 6.5 minutes, while it was 10.71 minutes with the baseline tool. Thus, BAST demanded 39% less time to perform these tasks.

![Average time spent on analysis of bug reports.](image)

(a) Duplicates avoided in the first step. (b) Time spent in the first step.

**Figure 3. Time spent and duplicates avoided in the first stage.**

### 3.1.2. Analysis of the Second Stage

During the second stage, it was analyzed 99 bug reports. The amount of analysis of bug reports increased considerably in this stage. One of the reasons for that was the increasing of the project also, i.e. number of features implemented and test cases. Figure 4 shows a graph with the classification of bug reports after the analysis conducted by the submitter during this stage. According to the chart, 44% of bug reports were duplicates that had been avoided.

![Respository status in second stage.](image)

**Figure 4. Respository status in second stage.**

**Number of duplicate bug reports avoided.** Figure 5(a) shows the percentage of duplicate bug reports that were avoided according to the tool used. In this second stage, the bug reports were analyzed first using BAST, and if it was not found similar bug reports, a new analysis should be performed using the baseline tool. At the end of this stage, the analysis made with the BAST avoided the submission of 89% of duplicates, while the
other tool avoided 7% that BAST could not avoid. In addition, 7% were avoided due to the fact that the submitter had prior knowledge of similar bug reports.

**Average time spent on analysis of bug reports.** According to the graph of Figure 5(b), the average time spent with search and analysis of bug reports with both tools did not diverge so much. However, BAST had the best performance in the second stage in regard to the time of analysis. That is, the submitter spent less time to analyze bug reports using BAST.

![Figure 5. Time spent and duplicates avoided in the second step.](image)

3.1.3. Analysis of the Whole Period

**Types of bug reports analyzed.** Figure 6 shows the types of bug reports that were analyzed in the study. As can be seen, 53% was composed of duplicates, which would enter in the repository if it would not avoided. Meanwhile, only 29% of bug reports addressed issues not yet submitted, and thus submitted to the repository. The other types of bug reports, although they were not duplicate, have not been submitted to the repository because they were invalid.

![Figure 6. Respository status.](image)

**Number of duplicate bug reports avoided.** As mentioned earlier, 53% of the bug reports were not submitted because they were duplicates. Figure 7(a) shows that 69% of those bug reports were avoided by using our tool, while only 28% have been avoided with the baseline tool. In addition, 3% of duplicates could only be avoided due to information provided by developers, or because the submitter already know that the bug report was duplicate. Thus, we can answer the first question positively, where BAST is more effective than the baseline tool.
Duplicates avoided.

Figure 7. Duplicates avoided and time spent for the whole period.

Average time spent on analysis of bug reports. Figure 7(b) shows that, in general, the average time spent to perform analysis of bug reports is reduced when using BAST. On average, BAST saves half the time it would be spent if the submitter was using the baseline tool. Therefore, it can be considered achieved the goal of reducing the time spent on analysis of bug reports. Thus, we can answer the second question positively, where BAST can reduce the time needed to search and analyze bug reports.

3.2. Confounding Factors

The results mentioned earlier showed that BAST can prevent more duplicates than the baseline tool, while also helps reducing the time spent on analysis of bug reports. However, a few factors of confusion need to be highlighted. The first factor refers to how many people were used to conduct the case study. As mentioned, only one submitter carried out the study. It is a factor of confusion because we cannot generalize the results to other submitters, with different experiences and background.

A second factor of confusion relates to the way that the tools were evaluated. The way the case study was designed, the submitter could only perform the search and analysis for the same bug report using both tools if no duplicates were found using the first tool of the current stage. This is a factor of confusion because we cannot compute the average time for the same search and analysis on different tools. In addition, if a duplicate was found using the first tool, it was not possible to verify if the same duplicate would also be found using the second tool.

We must also consider that the time to perform a second analysis can be influenced by the first analysis. For example, if the submitter spends a certain time to do an analysis in the first tool, but does not find any duplicate, it is likely that he will spend more time using the second tool to find a duplicate, if able. In this regard, we must also consider the factor of accommodation, in which the submitter tends to spend more time using the tool that he feels more comfortable to use.

4. Related Work

The first related work was from John Anvik [2]. He analyzed potential problems raised by bug repositories from Eclipse and Firefox projects, such as dynamic assignment of bug reports and bug report duplication. Although our work focused only in the duplication problem, we believe that it is a complement of Anvik’s work, since we expanded the number of projects and variables analyzed in order to understand the problem.
Hiew’s work [11] presented an approach to detect duplicate bug reports using clustering techniques. Runeson’s work [16] addressed the bug report duplication problem using Natural Language Processing (NLP) [10]. While Wang et al. [19] proposed an approach to detect duplicate bug reports using NLP and execution information. The execution information consists of data about software execution when an error occurs, such as method calls or variable state.

Another work to detect duplicate bug reports was performed by Jalbert and Weimer [12]. They proposed a system to automatically classify duplicate bug reports as they arrive. As a differential from the others work, the system used surface features and textual semantics in conjunction with clustering techniques.

Bettenburg et al. [4, 5] tried a different approach. Instead of avoiding duplicates, they propose to merge the duplicate bug reports to assist developers with additional information (such as *tracebacks*, or even descriptions from different points of view).

The work of Sandusky et al. [17] differs substantially from the previous ones. It proposed a method to identify and visualize bug report networks. With such networks, it was possible to analyze relationships and dependencies among bug reports.

5. Conclusion
In this work we described the problem of bug report duplication, which consists in submitting two or more bug reports concerning the same issue for one software project. Such problem is critical because, due to the presence of duplicates, it demands extra time to perform the search and analysis of bug reports during the submission phase.

Furthermore, we described a tool (BAST) developed to optimize the discovery of duplicate bug reporting and to decrease bug report analysis time. BAST was evaluated in a case study within a testing center of Motorola, and the results were very satisfactory. For example, the analysis performed with our tool reduced by half the time normally spent using the baseline tool (from Motorola) for search and analysis of bug reports. In addition, our tool prevented more than 60% of duplicate bug reports submission, if compared with the baseline tool.

For future work we are planning an experiment with 20 submitters to evaluate the gains of using BAST against well know tools (such as Bugzilla, Mantis and Trac) and inside different contexts. However, the design of the case study must be reviewed before running the experiment. That is because the current design turn difficult to measure which tool demands less time to perform search and analysis of bug reports. It is mainly because of the confounding factor of accommodation, as mentioned before. Thus, a better design to the case study would be to execute the search and analysis using only one tool in the first moment, and using only the other tool in the second moment.

Finally, the tool will also be evolved to move from the prototype stage to a more mature stage, and visualization techniques are being incorporated into it [9] with the objective of improving the amount of time spent and duplicates avoided.

Acknowledgement. This work was partially supported by the National Institute of Science and Technology for Software Engineering¹, funded by CNPq and FACEPE, grants 573964/2008-4, APQ-1037-1.03/08 and RHAE-559839/2009-0.

¹INES - http://www.ines.org.br
References