2013 17th European Conference on Software Maintenance and Reengineering

Finding Duplicates of Your Yet Unwritten Bug Report

Johannes Lerch
Technische Universität Darmstadt
Darmstadt, Germany
lerch@st.informatik.tu-darmstadt.de

Mira Mezini
Technische Universität Darmstadt
Darmstadt, Germany
mezini@st.informatik.tu-darmstadt.de

***Abstract*—Software projects often use bug-tracking tools to keep track of reported bugs and to provide a communication platform to discuss possible solutions or ways to reproduce failures. The goal is to reduce testing efforts for the development team. However, often, multiple bug reports are committed for the same bug, which, if not recognized as duplicates, can result in work done multiple times by the development team. Duplicate recognition is, in turn, tedious, requiring to examine large amounts of bug reports.**

Previous work addresses this problem by employing natural-language processing and text similarity measures to automate bug-report duplicate detection. The downside of these techniques is that, to be applicable, they require a reporting user to go through the time-consuming process of describing the problem, just to get informed that the bug is already known.

To address this problem, we propose an approach that only uses stack traces and their structure as input to machine-learning algorithms for detecting bug-report duplicates. The key advantage is that stack traces are available without a written bug report. Experiments on bug reports from the Eclipse project show that our approach performs as good as state-of-the-art techniques, but without requiring the whole text corpus of a bug report to be available.

***Keywords*-stack trace; duplicate detection; bug report**

I. Introduction

Duplicate bug reports arise when multiple users report problems caused by the same underlying fault multiple times because they are not aware that another report regarding the same fault has already been filed. Detecting duplicate bug reports is important for several reasons. First, it avoids that the work needed to fix a bug is repeated several times by different or the same developers. Second, an existing bug report describing the same problem can be helpful, even if that bug is not yet fixed, since the additional information given by another reporter can provide a different perspective of the same fault while adding helpful details and making the localization of faults easier [1].

Manual detection of duplicate bug reports puts a lot of burden on open-source projects, whose developers need to spend valuable time screening new and existing bug reports. Anvik et al. [2] assume that for each bug report at least 5 minutes of reading and handling are required. For the Eclipse project[1] with an average of 29 reports per day this would require an effort of two person-hours per day. We believe that the overhead is actually higher. Anvik et al. studied a data set between January and May 2005. On our dataset consisting of Eclipse bug reports in a larger time span (October 2001 to December 2007) the average of bug reports per day is at 93.

Many projects try to reduce this effort by delegating the task over to the users reporting new bugs. For example, the guidelines of the Eclipse project state: *"Search Bugzilla, to see whether your bug has already been reported"*[2] as a preliminary to filing a new report. However, this approach is moderately effective, as users are not willing to spend much time or are not experienced enough to be able to decide whether the same fault is described.

To simplify the task of finding duplicates most bug tracking tools provide search functionality that has been improved by using natural language processing [3]–[5] and execution traces [6]. Such search features require that the user reporting a bug writes the report, before being able to find out that someone else already reported that bug. This can be frustrating for reporters, eventually resulting in less users reporting bugs. To get an intuition of the amount of effort spent before a bug report is marked as duplicate, we counted the words written until a bug report in the Eclipse project is flagged as duplicate: on average 292 words written by the reporter and commenting users.

The insight driving the work presented in this paper is that duplicate bug detection may eventually rely on implicitly available information about the execution environment such as the operating system, the used runtime environment (e.g. *Java HotSpot(TM) Client VM (build 20.6-b01)*), the version of the crashed application, the installed plug-ins, the libraries and their version in the classpath, or simply stack traces written to a log file or available through an error handling mechanism, thus avoiding that the user has to explicitly describe the failure.

Specifically, in this paper we present an approach that relies on stack traces. Many applications already use error handling mechanisms that effortlessly make stack traces available to users. Moreover, a search for bug reports that is invoked automatically when a failure occurs, can be seamlessly

[1] http://eclipse.org

[2] https://bugs.eclipse.org/bugs/page.cgi?id=bug-writing.html

1534-5351/13 $26.00 © 2013 IEEE
DOI 10.1109/CSMR.2013.17

integrated within the error handling mechansims. Users can then be invited to file a new bug report, if the failure was not reported before and otherwise to complement existing reports. Yet, to be applicable, an approach based on stack traces requires that the latter are attached to bug reports. Most bug trackers, including the Bugzilla bug tracking software[3] used by the Eclipse project, do not come with attributes to store stack traces as part of a bug report. But, Bettenburg et al. [7] found that users work around this lack of explicit support by posting stack traces as part of descriptions and comments. Bettenburg et al. [7] also showed that it is possible to extract the stack traces contained therein for further processing.

The dataset used for our experiments contains 211,843 Bugzilla entries of the Eclipse project retrieved by Zimmermann for the MSR Mining Challenge 2008[4], of which 180,886 are bug reports, i.e., not tagged as enhancement[5]. Out of these bug reports 18,469 (roughly 10%) contain at least one stack trace that can be used for duplicate detection, i.e., it could be possible to effortlessly detect duplicates for these, while detection for others can still be done by existing techniques requiring text.

For the detection we propose an approach based on well-known search techniques. Namely, we use term frequency and inverse document frequency to rate stack traces. Our evaluation shows, that we can predict duplicates with a precision similar to usual text-based implementations. Moreover, we found that the text-based implementations themselves perform significantly better if bug reports contain stack traces.

To summarize, the work presented in this paper makes the following contributions:

- It suggests to detect duplicates before users have to write a bug report.
- It shows that previous approaches based on natural language processing significantly benefit from the existence of stack traces.
- It presents an approach that detects duplicate bug reports using stack traces only.
- It extensively evaluates that approach, also showing that stack trace based duplicate detection performs similar to existing text-based implementations.

The remainder of this paper is structured as follows: In Section II, we discuss how to extract stack traces from the textual parts of bug reports. In Section III we describe the structure of our search index. Section IV describes the search algorithm. We pick up the idea of time frames in V and evaluate our approach in Section VI, followed by a discussion of threats to validity in VII. Related work is discussed in Section VIII. Section IX summarizes this paper and outlines areas of future work.

[3]http://www.bugzilla.org

[4]http://msr.uwaterloo.ca/msr2008/challenge

[5]The Eclipse project uses its bug tracker not only to manage bug reports, but also feature/change requests, which are tagged as enhancements.

```
( [MODIFIER]? [EXCEPTION] ) ([:] [MESSAGE])?
( [at] [METHOD] [(] [FILE] [:] [LINE] [)] )*
```

Figure 1. Template for stack traces defined by Bettenburg et al. [7]

```
[EXCEPTION] ([:] [MESSAGE])?
( [at] [METHOD] [(] [SOURCE] [)] )+
( [Caused by:] [TEMPLATE] )?
```

Figure 2. Extended template for stack traces with recursive definition to match nested stack traces.

II. Detection of Stack Traces

Since most bug-tracking tools have no support for capturing stack traces, users typically embed stack traces in descriptions and comments. For building a search index for available bug reports, stack traces must thus be extracted from textual representations. Bettenburg et al. [7] provide the template in Figure 1 to describe the general structure of a stack trace in Java —the language we focus on in this work— which should make it easy to define regular expressions. Actually, this is a non-trivial task. While the overall structure of a stack trace is well defined, parts of a stack trace are not very specific; the message, e.g., is allowed to contain arbitrary content. In addition, stack traces are embedded in comments that can contain words that also occur in stack traces. Moreover, stack traces themselves can be cluttered with line breaks added by text formatters that enforce a maximal number of characters per line. These challenges render the detection as a trade-off between the rate of parts that are falsely assumed to be a stack trace and missed stack traces.

We adapted the template in Figure 1 in two ways. First, we added support for nested stack traces. Java allows caught exceptions to be re-thrown, whereby the exception thrown last contains a reference to the one before and so on. Nesting of exceptions results in nested stack traces. In our dataset 8% of all stack traces contain nested stack traces. We observed nesting up to a depth of 6. The second change is required as we observed that the definition of the content between the brackets of each frame does not necessarily follow the format `FILE:LINE`. Other values observed are `Native Method`, `Unknown Source`, and `ClassName.java(Compiled Code)`. Incorporating these additional requirements results in the template defined in Figure 2. To deal with arbitrary line breaks, we replace all newline characters by white spaces in a preprocessing step and adjust regular expressions to deal with white spaces at normally unexpected positions.

We define each part of the template as follows:
`EXCEPTION`: A fully qualified class name, thus containing word characters separated by `.` and `$` for nested classes. Also the character `/` is allowed as some virtual machines use them as package separator. To reduce false positives we add the

requirement that the class name must end with `Exception` or `Error` which holds for most exception types. In addition, we do not allow white spaces in this part, resulting in a small amount of stack traces not detected, but also reducing the number of false positives.

`MESSAGE`: Each thrown exception can have an optional message. No restrictions are imposed on the set of allowed characters. Consequently, for regular expressions it is required to use a reluctant rather than a greedy matcher to avoid matching consecutive frames.

`METHOD`: A fully qualified method, i.e., word characters separated by `.` or `/` and `$`. After the last separator, constructors `<init>` and static initializers `<clinit>` are allowed. We allow a limited amount of white spaces in method definitions, namely one allowed white space per 20 characters while multiple consecutive white spaces are counted as one. Our experiments have shown that this rate adequately separates typical text and methods containing white spaces generated by formatters.

`SOURCE`: Any characters are allowed as this content may differ on the logging framework or execution environment. This part can be different for otherwise identical stack traces without any meaning on the occurred failures; thus, it is left out for duplicate detection.

`TEMPLATE`: Recursive definition of the template to match nested stack traces. The string "*Caused by:*" is a reliable indicator for nested stack traces. If this indicator is missing a consecutive stack trace is treated as an independent stack trace.

We implemented the parser by splitting the defined template into multiple regular expressions, as even in that case it is not possible to parse all reports with the default Java stack-size setting, due to the recursion used in the Java regular expression implementation (with a maximum stack size of 10 MByte we could parse all bug reports without problems).

To verify that our parser implementation performs well on the dataset used for the evaluation we manually examined multiple samples and compared our results with the results of infoZilla, the tool implemented by Bettenburg et al. [7]. The parser of infoZilla finds 31,845 stack traces. We were able to handle some formatting issues not handled by their implementation and could detect 32,550 stack trace. Direct comparison of all detections was not possible due to the large manual effort, but random samples showed that in nearly all cases our implementation is more precise and has less false positives, while detecting slightly more stack traces.

III. Indexing

Common notions related to search indexes are documents and terms. A document is an element that a user wants to find; terms are arbitrary values that are linked with documents to make them findable, when users search for these terms.

In the duplicate-detection scenario a bug report can be considered as a document. By doing so, we would treat duplicate reports as additional and different documents to the original bug reports. Alternatively, building one document containing the original and all duplicates comes with the advantage that a problem may be indexed with synonymous terms, but also characteristic terms are easier to find when repeated in duplicates. When doing so, it is still possible to point users to the original bug report. All duplicates of that report can then be retrieved as they are linked by their duplicate marking. Therefore, we build a single document for an original bug report and all its duplicates. In the dataset used for evaluation we observed that bug reports are marked as duplicate of a bug report that is itself a duplicate of another bug report. We, therefore, include transitive duplicates in the indexed document. In the following, we will use the term *group* when referring to a bug report and its duplicates.

We employ the following indexed terms. The fully qualified exception type is used as a single term; any word contained in the message of length $>= 3$ is indexed in lower case; each fully qualified method of the call stack is considered as a single term. The source part is left out. Terms of nested stack traces are equally included, i.e. each stack trace can contribute multiple exception type terms. We use fully qualified names in normalized dot-separator representation.

IV. Searching

We use the term frequency and inverse document frequency approach - our implementation is based on Apache Lucene[6] and we stick with the default definitions to keep the implementation easily reproducible. A stack trace for which we want to find duplicates is tokenized in the same way as it would be tokenized when stored in the search index. To identify relevant documents, for each document in the index a score is computed; for performance reasons this is in fact only done for documents containing at least one matching term. The score function relates terms t in a query q with terms of an indexed document d as follows:

$$\text{score}(q,d) = \sum_{t \in q} \left[\text{tf}_d(t) * \text{idf}(t)^2 \right] \quad (1)$$

The score function makes use of term frequency (tf) and inverse document frequency (idf), which are defined as follows:

$$\text{tf}_d(t) = \sqrt{\text{frequency of } t \text{ in } d} \quad (2)$$

$$\text{idf}(t) = 1 + \log\left(\frac{\#\text{documents}}{\#\text{documents containing } t}\right) \quad (3)$$

Term frequency in (2) favors documents that frequently contain a term of the search query. To relativize this weight

[6]http://lucene.apache.org

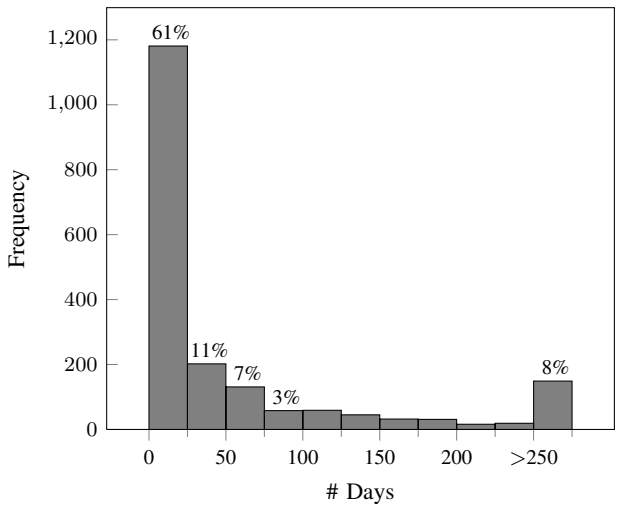

Figure 3. Histogram over days between the first and last bug report submission for each duplicate group.

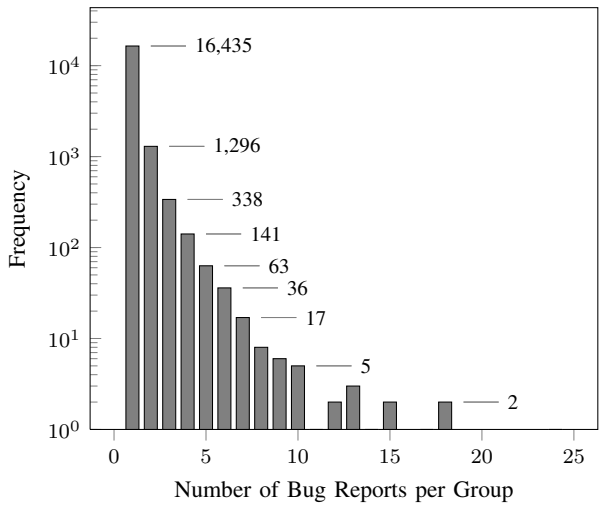

Figure 4. Histogram over the number of bug reports per group (only bug reports that contain stack traces are shown).

for terms that are used in most documents, inverse document frequency in (3) favors terms that are only used in a small amount of documents; thus, terms not characteristic for any problem get insignificant influence. The scoring function sums over all terms in the query so that documents matching more terms of the query get higher scores.

V. Time Frames

Runeson et al. [4] suggested that only reports that were submitted 50 days before the report for which duplicates should be detected are considered as potential duplicates. This improves scalability as the number of required comparisons is reduced. Furthermore, Runeson et al. found that by limiting the search space to a time frame also increases the recall rate of correctly identified duplicates. For their dataset, a time frame of 50 days works best and recall drops significantly (about 6%-8%), when the time frame is increased up to 500 days.

To estimate the applicability of time frames on our dataset, we measured the amount of days between the first and the last bug report submission in each group. Results are shown as histogram in Figure 3. Direct comparison with the measures gathered by Runeson et al. shows that in our dataset the percentage of groups with more than 100 days is greater (less than 1% for Runeson et al.). Also, on our dataset the average time span between bug reports is larger. Nevertheless, the amount of groups with ranges below 25 days is still high, so that time frames may result in higher recall rates on our dataset too. In Section VI-F, we will evaluate whether we can confirm that assumption.

VI. Evaluation

For evaluation, we simulate the submission of bug reports to a bug tracker based on real data. Each time a new report is submitted we will try to identify its duplicates that were already submitted to the bug tracker. For the detection only descriptions and comments written before the current simulated time can be used. This limitation ensures that only details are available that would be in a real usage scenario.

All comments and bug report submissions are inserted to the bug tracker in the order of their creation date. Before a new bug report is inserted to the bug tracker, it is checked whether another bug report of the same group is already inserted. If there is at least one, duplicate detection is triggered to predict the duplicate group. The duplicate detection does so by giving a ranked list of potential duplicate groups. The earlier the correct proposal is in the list, the better the results are.

To assess how well the proposed duplicate detection performs we compare it with a text based implementation described in Section VI-C. For the comparison we use metrics that are introduced in Section VI-B. But first, details about the used dataset are given.

A. Dataset

Our dataset consists of reports of the Eclipse project bug tracker[7] that was made publicly available for the MSR Mining Challenge 2008. This dataset contains 211,843 bug reports that were filed between October 2001 and December 2007. Out of these bug reports 30,957 are tagged as enhancement and represent feature/change requests. We will use the remaining 180,886 bug reports that describe bugs. These bug reports contain 32,550 stack traces located in 21,618 bug reports. 19,358 bug reports have stack traces in their problem description and are potential candidates for duplicate detection based on stack traces.

[7]https://bugs.eclipse.org/bugs/

In the evaluation all bug reports are inserted to the search index, but only bug reports that have duplicates already added to the bug tracker are used as query. We plotted the group sizes as histogram in Figure 4 to show the distribution of how many duplicates bug reports have. Note that for Figure 4 only bug reports are used that contain stack traces. 16,435 bug reports have no duplicates and the remaining 5,183 bug reports are distributed in 1,923 groups with more than one bug report each.

B. Metrics

To be able to compare the performance of the proposed approach we use two metrics: recall rate, which is also used by Runeson et al. [4], and mean reciprocal rank, which is often referred to as simplified variant of mean average precision. Recall rate is inspired by precision and recall, often used to evaluate information retrieval systems. In our evaluation scenario we will have a list proposed by the duplicate detection that contains either one correct reference to the duplicate group or none. Thus, precision will be $\frac{1}{\#\text{proposals}}$ or zero. Likewise, recall will be zero or one, respectively. As this would not be very meaningful, we adapt these metrics, so that recall rate is the overall percentage of proposals containing the correct reference. Obviously, the recall rate increases with the amount of proposals the duplicate detection is allowed to make. List sizes used in related work are 5 to 20 proposals, motivated by the assumption that users that report bugs are able and willing to scan these proposals for every bug they want to submit.

The second metric is mean reciprocal rank, which gives a single performance measure for arbitrary proposal list sizes w.r.t. the rank of the correct proposal. Mean reciprocal rank (MRR) is defined as

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i} \tag{4}$$

where Q is the set of bug reports for which duplicates should be proposed and $rank_i$ is the rank of the correct proposal made for i-th bug report. If the list of proposals does not contain the correct reference, the second fraction is assumed to be zero.

C. Baseline implementation

To compare our stack-trace-based approach to duplicate detection with an approach using a bug report's full text, we implemented a conventional text-based search approach (baseline). Based on Apache Lucene, we implemented the approach using tokenization on all non-word characters, transforming characters to lower case, and dropping words with less than four characters. Figure 5 shows the recall rate of the baseline approach in relation to the used list size for different parts of the data set.

The text-based approach we use as the baseline corresponds to conventional algorithms used in today's bug trackers.

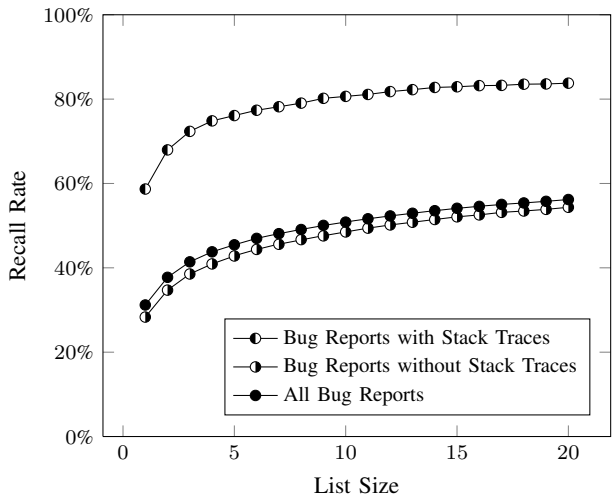

Figure 5. Recall rate of the baseline implementation on bug reports containing stack traces and bug reports without stack traces.

We are aware that previous work [4]–[6] has shown that better results can be achieved by stemming, synonym words, stop word removal and further natural language processing techniques. A comparison of the reported recall rates of the respective publications with the recall rates of our baseline implementation indicates that these more sophisticated techniques would outperform our baseline duplicate detection by 3%-8%. As we will explain in Section VI-D, this is negligible for the purpose of our evaluation.

A much more interesting observation revealed by the graphs in Figure 5 is that our text-based baseline performs much better on bug reports containing stack traces: recall rates are 30%-35% higher on bug reports that contain stack traces than on bug reports without stack traces. Obviously, this observation carries over to other more sophisticated text-based approaches.

D. Stack Trace Duplicate Detection

In this subsection, we evaluate how an approach based solely on stack traces performs compared to the baseline. To keep a level playing field, both approaches are given the same bug reports to create indexes and to generate duplicate proposals for. The results are therefore directly comparable, as only the features used of a bug report are different. We compare three settings: (a) the proposed approach based on stack traces, i.e., only stack traces are put in the index and used when searching for duplicates, (b) the baseline implementation indexing text and using the whole description of a bug report to find potential duplicates, and (c) a hybrid approach that basically is the baseline implementation, but which uses only stack traces as query. This last one is comparable to applying conventional duplicate detection mechanisms to stack traces.

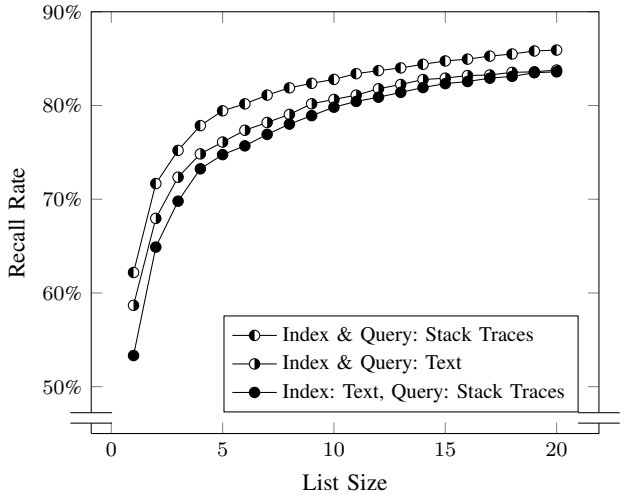

Figure 6. Recall rate of the stack trace based duplicate detection compared to text based approaches.

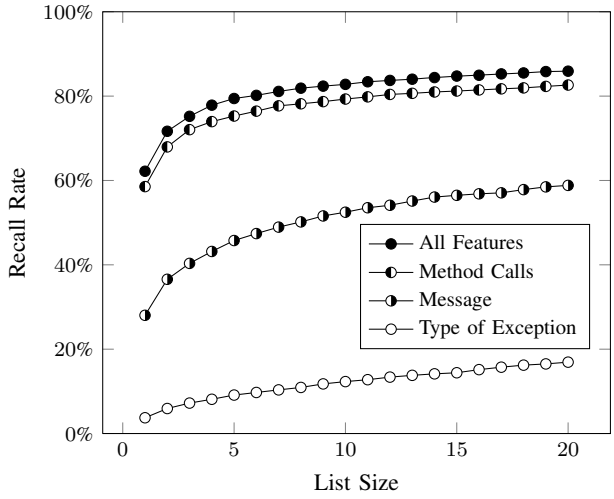

Figure 7. Recall rate of the stack trace based duplicate detection while using only parts of the features available in a stack trace.

The recall rates for the three settings are shown in Figure 6 The proposed duplicate approach based on stack traces alone is shown as *"Index & Query: Stack Traces"*; the baseline implementation is *"Index & Query: Text"*. The results of the hybrid approach are shown as *"Index: Text, Query: Stack Traces"*. The results show that the stack-trace-based approach performs slightly better than the baseline, despite the fact that it uses less information. This confirms that stack traces are valuable information for duplicate detection and can be used without having text-based details of a bug report. The hybrid approach achieves about 5% lower recall rate compared to the approach optimized for stack traces for a list size up to 5. This gap can be explained by the fact that in the hybrid approach tokenization is adjusted for words of text, rather than the more structured elements of a stack trace. Also, the text used to create the index contains words that misleadingly may match parts of stack traces.

To summarize, we conclude that duplicate detection based only on stack traces is feasible and has comparable (slightly better) performance with the baseline approach that uses the whole text of a bug report. Compared to more sophisticated approaches than the baseline implementation we used, our approach is expected to perform the same or only slightly worse (basically, it will loose the advantage it has compared to the baseline implementation), while requiring much less information to be available, relieving users from the burden of having to write bug reports before being able to search for existing duplicates.

E. Detailed Analysis of Stack Trace Duplicate Detection

We performed further experiments to find out which parts of a stack trace are most important for duplicate detection. The results are shown in Figure 7. The plots indicate that the type of the thrown exception yields the lowest recall rates. Using the textual message results in better recall than using the exception type. Unfortunately, a message is not available in every stack trace, as it is an optional attribute for many exception types. In our dataset only 19,257 of 32,550 stack traces contained a message. Accordingly, we performed the experiment for the setting that involves only the message on bug reports with stack traces that do contain a message. Of all elements of a stack trace, method calls are the most effective means for duplicate detection. This means that the error message written by developers is less effective than the knowledge about the methods that were called prior to the failure. It seems that most of the reported bugs in our dataset were errors not expected by the developers; any defined error handling mechanism may be too general and not able to describe the problem precisely.

We performed further experiments to find out whether method calls in certain stack trace positions are more effective than others w.r.t. classifying new bug reports into existing report groups (e.g., are the last ten method calls before the exception was thrown more important than the rest?). To answer this question, we define D_p below as a means to characterize the impact of method calls at stack trace position p on the classification of bug reports into report groups. The definition of D_p uses the metric $C_g(m)$ shown in (6), which quantifies the degree to which the occurrence of a method m on the stack trace is characteristic for a bug report group G and builds the average of $C_g(m)$ over all stack traces S and the called methods at each position $M_{p,s}$:

$$D_p = \frac{1}{|S|} \sum_{s \in S} \left[\frac{1}{|M_{p,s}|} \sum_{m \in M_{p,s}} C_{\text{group}(s)}(m) \right] \quad (5)$$

$$C_g(m) = \frac{\text{frequency of } m \text{ in group } g}{\text{total frequency of } m} \quad (6)$$

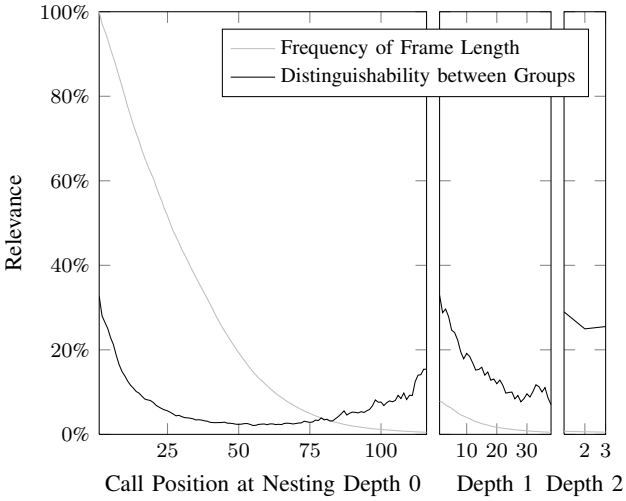

Figure 8. Frequency of frame length and uniqueness of methods that were observed at specific frame positions.

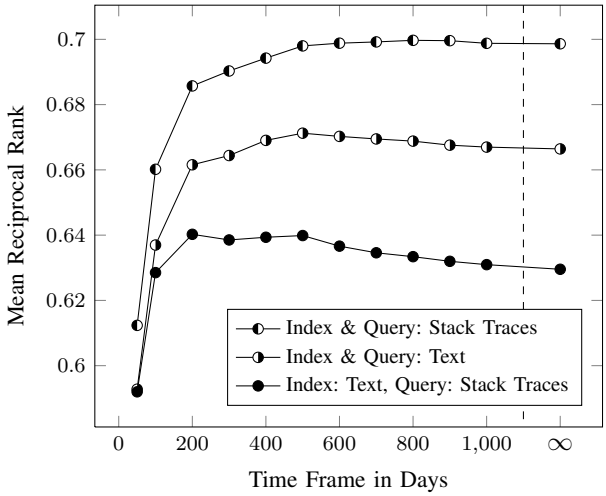

Figure 9. Mean Reciprocal Rank for different time frames on bug reports containing stack traces.

The results of our experiments are shown in Figure 8. The figure is split in three parts. The left part considers only outmost stack traces; the middle part considers the first nested stack trace (depth 1); the right part considers nested stack traces of depth 2. Each part displays two lines.

The light gray lines show the frequency of calls at certain positions in our dataset. For example, calls at position 50 have a frequency of roughly 20%, meaning that 20% of the stack traces in the dataset have at least a length of 50 stack frames. Nested stack traces of depth 1 are available in 8% of all stack traces, as can be seen at the frequency of the first call in the middle part. Nesting of depth 2 occurs with a frequency of 0.7%, i.e., it is very rare.

The dark gray lines visually depict D_p for each P in the x-axis. Obviously no single position is capable of classifying reports to a duplicate group with high certainty, i.e., none of the positions has a significant impact on the classification. It is also evident that the first 20 calls have a higher impact than the following 70 calls. The impact of positions higher than 90 increases again. However, the fraction of stack traces having more than 90 calls is below 5% (light gray line). The first calls of the nested stack traces again have higher impact.

Based on these findings we recommend that users always post complete stack traces to bug reports. As complete stack traces require a lot of space in a bug report, thereby making the description hard to read, we assume that many users shorten stack traces. To avoid this behavior, we propose to use additional fields allowing stack traces as attachments to bug reports, such that they can be displayed independently of descriptions.

F. Time Frames

Runeson et al. [4] were able to improve the recall rates of their text-based duplicate detection algorithm by limiting the amount of potential duplicates considered to a fixed time frame before the submission of a new report. They evaluated their technique on a non-public dataset containing bug reports of Sony Ericsson. Our analysis of the Eclipse dataset in Figure 3 showed that time spans between bug reports marked as duplicates are larger than for Sony Ericsson. This first impression is confirmed by our experiments. In Figure 9 the same measurements as before are shown, but this time mean reciprocal rank in relation to time frame size is shown. Runeson et al. stated that for Sony Ericsson a time frame of 50 days works best. For Eclipse 50 days performs significantly worse (3%-8% less) than without a time frame. Best results for Eclipse are between 200 and 800 days, depending on the used approach. Interestingly, the text based approaches perform best with smaller time frames (200 days for *"Index: Text, Query: Stack Traces"* and 500 days for *"Index & Query: Text"*) than the stack traces only approach (800 days). Also, the mean reciprocal rank remains relatively stable when increasing time frames further for stack traces only, while values for text based approaches drop. Overall, time frames were not able to increase performance of the stack-trace-based approach significantly and have only minor influence on the text-based approach.

We therefore cannot confirm that improvement of recall rates by using time frames generalizes to our dataset and/or our stack-based detection. In any case, time frames have to be adjusted for each project, otherwise recall rates may drop significantly.

VII. Threats to Validity

Several assumptions underlay the validity of our experiments and applicability to real usage scenarios. Our setup assumes that users reporting bugs are willing and experienced enough to identify duplicates in a list of existing bug reports

if suggested to them. Even though we reduce the effort in duplicate detection by not requiring users to write the report before able to find duplicates, users may still not have the time to read and understand existing bug reports.

Another threat may be due to the used dataset itself. As the dataset consists of real bug reports duplicate markings may be wrong or missing. The bug reports were written and maintained by hundreds of people that can't overview the whole amount of bug reports, thus probably miss some duplicates. Wrong usage of the bug tracker may also lead to missing duplicate markings, when bug reports are just closed instead of being marked as duplicate.

The bug reports of the used dataset were created while no automated duplicate detection was enabled in the bug tracker. On one hand this ensures that the dataset does not have any bias induced by another approach to duplicate detection, on the other hand the use of an automated duplicate detection combined with manual examination may could have lead to a more complete dataset.

As in every empirical study our findings may not be generalizable to other projects. Other projects may use their bug tracker in different ways and provide data within a bug report that differs from typical data in Eclipse bug reports. As our approach depends on stack traces, the used frameworks, libraries, and the software architecture of the project itself could have large influences on the stack traces generated on software failures.

VIII. Related Work

Multiple publications inspecting characteristics of bug reports and duplicates exist.

A. Meta Studies

In 2008, Bettenburg et al. [7] implemented infoZilla, a tool to extract structural information from bug reports. Beside code attachments and steps to reproduce they also extracted stack traces. We compared our implementation of a stack trace parser with theirs. Manual inspection of samples showed that we could reduce false positives and increase the overall detection rate.

Bettenburg et al. [1] also found that duplicate bug reports often contain valuable additional information that can help reproduce and fix bugs. They suggest to rethink submission guidelines that state duplicates as harmful; instead, duplicates should be seen as a non-ideal representation of additional information.

Bettenburg et al. [8] consulted developers and bug reporters to ask, what is usually provided by the latter and what is most useful to the former. Their results show that stack traces are one of the most useful items for developers and that they should be provided more often than is current practice. Schröter et al. [9] investigated the importance of stack traces in the bug-fixing process and were able to provide statistical evidence for the statements developers made during the survey of Bettenburg et al.: up to 60% bug reports containing stack traces have fixes changing code in a method part of a posted stack trace. Usually, those changes were made in one of the top-10 stack frames. Another result is the finding that bug reports containing stack traces get fixed faster. Our experiments supplement those findings, as we found that stack traces are also a reliable factor when detecting duplicates.

In 2004, Cubranic and Murphy [10] investigate the problem, who can best fix a bug. They trained a naive bayes classifier on text of bug reports to predict assignments of developers to bug reports and evaluated the classifier on 15,859 bug reports of the Eclipse project. On 30% of the reports they were able to predict the correct assignment. Experimenting with multiple machine learning algorithms, Anvik et al. [11] were able to increase the rate of correct predictions to 57% for bug reports of Eclipse and 64% for Firefox. This was possible by using more features of a bug report than just the description and summary fields, by preprocessing data, and by the use of Support Vector Machines.

Rastkar et al. [12] automatically generate summaries of bug reports, making them easier to comprehend and thus easier to detect as duplicates manually. They generate summaries by classifying each sentence of a bug report as representative or not. For experiments they trained a classifier on manually created summaries.

B. Duplicate Detection

In 2005, Anvik et al. [2] liken duplicate detection to the detection of spam mails and state that when most spam can be filtered out, filtering duplicates should be possible, too. They built a statistical model and constantly update it on each submitted report. Using cosine similarity and the statistical model they were able to detect 28% of all duplicate bug reports in the Firefox bug tracker.

Hiew [3] reached 29% precision and 50% recall on the Firefox bug tracker and for Eclipse 14% and 20%, respectively. He does so by incremental clustering, i.e. by representing similar bug reports by centroids and comparing new bug reports with each centroid. When similarity exceeds a specified threshold, the new bug report is treated as duplicate otherwise unique.

Runeson et al. [4] detect duplicates using natural language processing, i.e., using tokenization, stemming, stop words removal, synonyms and spell checking. They found that limiting the search space to time frames increases recall rates by 6%-8%. They evaluated their approach on a dataset consisting of non-public bug reports of Sony Ericsson. For a suggestion list size of 5 they reach about 31% recall rate that increases to 42% for a list size of 15. They did interviews with testers and analysts. One tester said that any tool making it easier to find duplicates is good, but also that it would be

hard to get testers to use the tool in the intended way as it requires to first enter a full bug report.

Wang et al. [6] also used natural language processing, but combined it with using execution traces. With the additional information they were able to reach recall rates of 67%-93% for suggestion list sizes 1-10. They had to acquire execution traces manually by reproducing steps described in bug reports as they are not existent in the bug reports of Eclipse and Firefox they used.

Jalbert and Weimer [13] propose a classification approach that predicts whether a submitted bug report is a duplicate or not. The approach is based on a combination of surface features, textual similarity metrics and graph clustering. Experiments on 29,000 bug reports of the Mozilla project showed that their approach is able to filter out 8% of the duplicate bug reports while allowing at least one report per defect to reach developers.

Sureka and Jalote [14] used a n-gram-based approach for duplicate detection, thus their approach is independent of the used language in bug reports. Experiments were done on the same Eclipse dataset we used and they were able to reach a recall rate of 33.93% for a suggestion list size of 50.

Sun et al. trained Support Vector Machines to calculate the probability if two given bug reports are duplicates of each other [15]. They provided an improved approach based on an adaption of BM25F [16] that reaches recall rates of 37% to 71% for suggestion list sizes of 1 to 20.

Prifti et al. [5] investigated the needs of duplicate detection when applied to large amounts of bug reports. They found that typical approaches have decreasing recall rates with increasing amounts of bug reports. Their solution to this problem is the use of time frames, which they showed in experiments on 74,585 bug reports of Firefox are applicable to keep recall rates constant.

Kaushik and Tahvildari [17] investigated on the performance of different information retrieval methods and heuristics for duplicate detection and compared these in an evaluation based on bug reports of the Eclipse (4330 bug reports) and firefox (9474 bug reports) projects. Following the finding of Schröter et al. [9], they included the identifiers of the 10 top most stack frames in the duplicate detection process. For Eclipse this improved recall rates. While this is an interesting effect that we also found, they didn't investigate the applicability of using stack traces alone and lack descriptions of how to extract and integrate them.

The main difference between all those approaches and ours is that they assume a bug report to be written before applying techniques to detect duplicates.

C. Bucketing Crash Reports

Software development companies using automatic crash report collectors for their programs face the problem of having a lot of crash reports describing the same problem. To prioritize efforts spend on those it's important to know how often a failure occurred, thus crash reports describing the same failure should be placed in the same bucket. In this area multiple papers are published to solve the problem in which bucket a crash report should be placed. Although that crash reports contain different data than bug reports, as they are generated automatically, the techniques used are based mainly on call stacks and thus related to our work.

Brodie et al. [18] and Modani et al. [19] tackle the problem by adapting stop-word removal to call stacks, removing recursive calls, and using similarity measures like edit distance, longest common subsequence, and prefix matching. Bartz et al. [20] also used edit distance, but differentiate between seven types of edits that they assign with different weights. Kim et al. [21] aggregate multiple call stacks of one bucket as crash graphs, making it easier for developers to identify frames that may cause the failure. They also showed that crash graphs can be used to detect duplicate crash reports. Dang et al. [22] introduced a clustering for crash reports using a new similarity measure called Position Dependent Model that uses the position of a function in the call stack and the offset between matched functions.

The listed approaches work on C based call stacks, thus findings are not necessarily generalizable for Java stack traces we focused on. Experimental results are also not comparable as in the scenario of crash reports the amount of duplicates is much higher, as data is collected automatically while in the bug report tracking scenario most bug reports are written manually.

IX. Conclusion

In this paper, we presented an approach to duplicate detection that doesn't require the reporter to waste time on writing a duplicate bug report. This is achieved by using the stack trace automatically generated during a software crash.

Experiments on 21,618 real bug reports of the Eclipse project showed that the proposed approach performs slightly better than conventional approaches without requiring the a-priori availability of the complete text of a bug report. We also investigated the influences of time frames proposed by Runeson et al. [4] and found that the improvements in recall rate they observed are not reproducible on the dataset we used for experiments. Also, the influence of time frames is lower for the stack-trace-based approach than for text-based approaches.

Like other studies that showed the importance of having stack traces in bug reports [8], [9] we strongly recommend to add support for stack traces in all bug trackers. Without support for stack traces in bug trackers, there is always the danger that stack traces get shortened and important parts are left out just to keep the report readable without having to scroll down many pages. Our experiments showed that the parts at the bottom of a printed stack trace, calls and nested stack traces, still have influences when determining whether

the stack trace describes a problem that is already reported. Unfortunately, those are the first which users typically leave out when shortening stack traces.

Future works will investigate the applicability of other a-priori available details discussed in section I for duplicate detection. We also plan to elaborate using stack traces as search queries for help and documentation platforms that could enable automatic pointing to helpful resources each time a stack trace occurs during development. Going further, the assumption that similar stack traces describe the same problem may allow inference of how to fix a bug knowing how previous bugs got fixed that had a similar stack trace.

ACKNOWLEDGMENT

We thank Eric Bodden and Marcel Bruch for sharing and discussing ideas on this topic, Andreas Sewe and Ben Hermann for their helpful comments on drafts of this paper, and Nicolas Bettenburg for sharing the infoZilla parser implementation.
This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE.

REFERENCES

[1] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful ... really?" in *International Conference on Software Maintenance*, 2008, pp. 337–345.

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange*, 2005, pp. 35–39.

[3] L. Hiew, "Assisted detection of duplicate bug reports," Masterthesis, UBC, 2006.

[4] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th international conference on Software Engineering*, 2007, pp. 499–510.

[5] T. Prifti, S. Banerjee, and B. Cukic, "Detecting bug duplicate reports through local references," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering, PROMISE*, 2011, pp. 8:1–8:9.

[6] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 461–470.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 27–30.

[8] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 308–318.

[9] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010, pp. 118–121.

[10] D. Čubranić, G. Murphy, F. Maurer, and G. Ruhe, "Automatic bug triage using text categorization," in *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering*, 2004, pp. 92–97.

[11] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 361–370.

[12] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, 2010, pp. 505–514.

[13] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Dependable Systems and Networks (DSN)*, 2008, pp. 52–61.

[14] A. Sureka and P. Jalote, "Detecting duplicate bug report using character N-gram-based features," in *17th Asia Pacific Software Engineering Conference, APSEC*, 2010, pp. 366–374.

[15] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *29th International Conference on Software Engineering (ICSE)*, 2010, pp. 45–54.

[16] C. Sun, D. LO, S.-C. Khoo, and J. JIANG, "Towards more accurate retrieval of duplicate bug reports," in *Automated Software Engineering (ASE)*, 2011, pp. 253–262.

[17] N. Kaushik and L. Tahvildari, "A comparative study of the performance of IR models on duplicate bug detection," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012*, 2012, pp. 159–168.

[18] M. Brodie, S. Ma, G. M. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn, "Quickly finding known software problems via automated symptom matching," in *Autonomic Computing (ICAC)*, 2005, pp. 101–110.

[19] N. Modani, R. Gupta, G. M. Lohman, T. F. Syeda-Mahmood, and L. Mignet, "Automatically identifying known software problems," in *International Conference on Data Engineering Workshop (ICDE)*, 2007, pp. 433–441.

[20] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle, "Finding similar failures using callstack similarity," in *3rd Workshop on Tackling Computer Systems Problems with Machine Learning Techniques, SysML*, 2008.

[21] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *Dependable Systems and Networks (DSN)*, 2011, pp. 486–493.

[22] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1084–1093.