Locating Bugs without Looking Back

Tezcan Dilshener Michel Wermelinger Yijun Yu
Computing and Communications Department, The Open University, United Kingdom

ABSTRACT

Bug localisation is a core program comprehension task in software maintenance: given the observation of a bug, where is it located in the source code files? Information retrieval (IR) approaches see a bug report as the query, and the source code files as the documents to be retrieved, ranked by relevance. Such approaches have the advantage of not requiring expensive static or dynamic analysis of the code. However, most of state-of-the-art IR approaches rely on project history, in particular previously fixed bugs and previous versions of the source code. We present a novel approach that directly scores each current file against the given report, thus not requiring past code and reports. The scoring is based on heuristics identified through manual inspection of a small set of bug reports. We compare our approach to five others, using their own five metrics on their own six open source projects. Out of 30 performance indicators, we improve 28. For example, on average we find one or more affected files in the top 10 ranked files for 77% of the bug reports. These results show the applicability of our approach to software projects without history.

1. INTRODUCTION

During software maintenance, a programmer can easily introduce semantic bugs due to inconsistent understanding of the requirements or intentions of the original developers. Previous studies performed with two open source software (OSS) projects showed that 81% of all bugs in Mozilla and 87% of those in Apache are semantics related [7]. These percentages increase as the applications mature, and they have direct impact on system availability, contributing to 43% to 44% of crashes. Since it takes a longer time to locate and fix semantic bugs, more effort needs to be put into helping developers locate the bugs, especially because a bug is often located in just a few of the thousands of files comprising an application. Lucia *et al.* [9] looked at 384 bugs from Rhino, AspectJ and Lucene and found that over 84% of them are located in just one or two files.

Recent empirical studies provide evidence that many terms used in a bug report (BR) are also present in the source code files [3]. Such BR terms are an exact or partial match of code constructs (i.e. file name, method name, variable or comment) in at least one of the files **affected** by the BR, i.e. those files actually changed to address the BR. It is claimed in [3] that although file names are typically a combination of 2-4 terms, they are present in more than 35% of the BR summaries and 85% of the BR descriptions of the OSS project AspectJ. Furthermore, BRs for OSS project Eclipse also contain stack trace information, and in 60% of BRs one of the affected files was mentioned in the stack trace [8].

MSR'16, May 14–15, 2016, Austin, TX, USA
 ISBN 978-1-4503-4186-8/16/05…$15.00.
DOI: http://dx.doi.org/10.1145/2901739.2901775

Our aim, motivated by these insights, is to leverage the occurrence of file names in BRs for IR-based bug localisation for Java programs. Current state-of-the-art approaches (BugLocator [2], BLUiR [3], AmaLgam [4] and LearnToRank [5]) rely on project history to improve the suggestion of relevant source files. In particular they use similar BRs and recently modified files. The rationale for the former is that if a new BR *a* is similar to a previously closed BR *b*, the files affected by *b* may also be relevant for *a*. The rationale for the latter is that recent changes to a file may have led to the reported bug. However, the observed improvements using the history information have been small. We thus wonder whether file names mentioned in the BR descriptions can replace the contribution of historical information in achieving comparable performance.

RQ*: Can the occurrence of file names in BRs be leveraged to replace project history in achieving state-of-the-art IR-based bug localisation?*

To address the RQ, we propose a novel approach, implemented in our tool *ConCodeSe* (Contextual Code Search Engine) [10], and then evaluate it against existing approaches [2, 3, 4, 5, 6] on the same datasets and with the same performance metrics. Like other approaches, ours scores each file against a given BR and then ranks the files in descending order of score. A bug (or BR) is considered **located** if at least one of the files affected by the BR is among the top-ranked ones, so that it can serve as an entry point to navigate the code and find the other affected files. As we shall see, our approach outperforms the existing ones in the majority of cases. In particular it succeeds in placing an affected file among the top-1, top-5 and top-10 files for 49%, 71% and 77% of BRs, on average.

2. PREVIOUS APPROACHES

Zhou *et al.* [2] proposed an approach consisting of the four traditional IR steps (corpus creation, indexing, query construction, retrieval & ranking) but using a revised Vector Space Model (rVSM) to score each source code file against the given BR. In addition, each file gets a similarity score (SimiScore) based on whether the file was affected by one or more closed BRs similar to the given BR. Similarity between BRs is computed using VSM and combined with rVSM into a final score, which is then used to rank files from the highest to the lowest. The approach, implemented in a tool called BugLocator, was evaluated using over 3,400 BRs and their known affected files from four OSS projects (see Table 1): the IDE tool Eclipse, the aspect-oriented programming library AspectJ, the GUI library SWT and the bar-code tool ZXing. Eclipse and AspectJ are well-known large scale applications used in many empirical research studies for evaluating various IR models [14]. SWT is a subproject of Eclipse and ZXing is an Android project.

Saha *et al.* [3] presented BLUiR, which leverages the structures inside a bug report and a source code file. To measure the similarity between a bug report and a file, the approach uses the Indri tf/idf model and incorporates structural IR. Indri computes a similarity score between each of the 2 fields (**summary** and **description**) of a bug report and each of the 4 parts of a source file (class, method, variable names, and comments). The 8 scores

are summed into a final score to rank the files. The results were evaluated using BugLocator's dataset and performance indicators. For all but one indicator for one project, ZXing's mean average precision (**MAP**), BLUiR matches or outperforms BugLocator, hinting that a different IR approach might compensate for the lack of history information, namely the previously closed similar BRs.

Table 1. Project Artifacts

Project	Source files	BRs	BR Period
AspectJ	6485	286	2002/07 – 2006/10
Eclipse	12863	3075	2004/10 – 2011/03
SWT	484	98	2004/10 – 2010/04
ZXing	391	20	2010/03 – 2010/09
Tomcat	2038	1056	2002/07 – 2014/01
ArgoUML	1685	91	2002/01 – 2006/07

Wang *et al.* [4] proposed AmaLgam for suggesting relevant buggy source files by combining BugLocator's SimiScore and BLUiR's structured retrieval into a single score using a weight factor, which is then combined (using a different weight) with a version history score that considers the number of bug fixing commits that touch a file in the past *k* days. This final combined score is used to rank the file. The approach is evaluated in the same way as BugLocator and BLUiR, for various values of *k*. AmaLgam matches or outperforms the other two in all indicators except one, again the MAP for ZXing.

Moreno *et al.* [6] presented LOBSTER, which leverages the stack trace (**ST**) information available in bug reports to suggest relevant source files. The approach first calculates a textual similarity score between the words extracted by the Lucene tool, which uses VSM, from bug reports and files. Second, a structural similarity score is calculated between each file and the file names extracted from the ST found in the bug report. If the file is not in the ST then the application's call-graph information is used to check whether a neighbouring file name occurs in the ST. Finally, both textual and structural similarity scores are combined to rank the files. The authors conclude that considering STs does improve the performance with respect to only using VSM. We also leverage ST and compare the performance of both tools using their dataset for ArgoUML, a UML diagraming tool (Table 1).

Ye *et al.* [5] defined a ranking model that combines six features measuring the relationship between bug reports and source files, using a learning-to-rank (LtR) technique: (1) lexical similarity between bug reports and files; (2) API documentation of the libraries used by the source code; (3) similar bug reports that got fixed previously; (4) bug fixing recency, i.e. time of last fix in terms of months; (5) bug fixing frequency, i.e. how often a file got fixed; (6) feature scaling, used to bring the score of all previous features into one scale. Their experimental evaluations show that the approach places the relevant files within the top-10 recommendations for over 70% of the bug reports of Tomcat (Table 1). Our approach is much more light weight: it only uses the first of Ye *et al.*'s six features, lexical similarity, and yet provides better results on Tomcat.

3. OUR APPROACH

ConCodeSe utilises state of the art data extraction, persistence and search APIs. The Java code is parsed using the source code mining tool JIM [11]. We also developed a Java module using the Lucene's [13] *Standard-Analyzer* to tokenise the text in the BRs into terms, which also includes stop-word removal. We use a publicly available stop-words list to filter them out [12].

Given a BR and a file, our approach computes two kinds of scores for the file: a **probabilistic score**, given by VSM as implemented by Lucene, and a **lexical similarity score**. Each kind of scoring is obtained with four search types using a different set of terms indexed from the BR and the file:

1. Full terms from the BR and from the file's code.
2. Full terms from the BR and the file's code and comments.
3. Stemmed terms from the BR and the file's code.
4. Stemmed terms from the BR, the file's code and comments.

For each of the 8 combinations of scoring, all files are ranked in descending order. Files with the same score are ordered alphabetically. Then, for each file we take the best of its 8 ranks. Again, files that have the same best rank are sorted alphabetically, by fully-qualified names, to get a total order that unambiguously defines the top-N ranked files.

The rationale for this approach is that during experiments, we noticed that when a file could not be ranked among the top-10 using the full terms and the code, it was often enough to use associated comments and/or stemming. As shown in Table 2, for SWT's BR #100040, the affected Menu.java file had a low rank (484th) using the first type of search with lexical similarity scoring. When the search includes comments, stemming or both, it is ranked at 3rd, 29th and 2nd place respectively. The latter is the best rank (in bold) for this file and thus returned by ConCodeSe (4th column of the table).

Table 2. Lexical similarity rank for all search types in SWT

BR #	Affected Java file	Bug-Locator	Con-CodeSe	1: Basic	2: Com-ments	3: Stems	4: 2+3
100040	Menu	20	**2**	484	3	29	**2**
79107	Combo	6	**3**	26	29	**3**	8
84911	FileDialog	6	**5**	**5**	39	6	56
92757	StyledText-Listener	87	**3**	4	**3**	75	72

There are cases when using comments or stemming could deteriorate the ranking, for example because it helps irrelevant files to match more terms with a BR and thus push affected files down the rankings. For example, in BR #92757, the affected file StyledTextListener.java is ranked much lower (75th and 72nd) when stemming is applied. However, by taking the best of the ranks, ConCodeSe can cope with such variations.

After the ranking and probabilistic scoring, we next explain the lexical similarity score, which is calculated for each source code file as follows.

(1) Check if the file name matches one of the words in key positions (**KP**) of the BR's summary, and assign a score.

(2) If no score was assigned and if a ST is available, check if the file name matches one of the file names listed in the ST and assign a score accordingly.

(3) If there is still no score then assign a score based on the occurrence of the BR's text terms (**TT**) in the file.

1) Scoring with key positions (KP score)

By manually analysing all SWT and AspectJ BRs and 50 random Eclipse BRs, i.e. (286+98+50)/4626=9.4% of all BRs (Table 1), we found that the word in first, second, penultimate or last position of the BR summary is likely to correspond to the affected file name. For example, for SWT, in 42% (42/98) of the BRs the

first word and in 15% (15/98) of the BRs the last word of the summary sentence was the affected source file.

Based on this, we assign a high score to a source file when its name matches a word in the four key positions of the BR summary sentence. The earlier the file name occurs, the higher the score: the word in first position gets a score of 10, the second 8, the penultimate 6 and the last 4. Disregarding other positions in the summary prevents non-affected files that occur in such positions from getting a high KP score and thus a higher rank.

2) Scoring with stack traces (ST score)

ST lists the files that were executed when an error condition occurs. During manual analysis of the same BRs as for the KP scoring, we found that many BR description fields include ST information. Especially for NullPointerException, the affected file was often the first one listed in the ST. For other exceptions however, the affected file was likely the second or the fourth item on the ST. Based on these patterns, and as for the KP scoring, to reduce false positives we only consider the initial positions and give a higher score to earlier occurrences of the file name.

We first use regular expressions to extract from the ST the application-only source files, i.e. excluding third party and Java library files, and put them into a list in the order in which they appeared in the trace. We score a file if its name matches one of the first four files occurring in the list. The file in the first position gets a score of 9, the second 7, the third 5 and the fourth 3.

3) Scoring with text terms (TT score)

We assign a score to the source file based on where the BR's terms (a set, without duplicates) occur in the file. If a BR term occurs in the file name this results in a higher score. Each occurrence of each BR term in the file increments slightly the score (Fig.1). As explained before, the BR's and the file's terms depend on whether stemming and/or comments are considered.

```
Algorithm: scoreWithFileTerms
input: file: File,   br: BR
output: score: float
begin
  score := 0
  for each query_term in br.terms
    if (query_term = file.name) return score + 2
    if (file.name contains query_term)
       score := score + 0.025
    else
      for each doc_term in file.terms
         if doc_term = query_term
           score := score + 0.0125
  return score
end
```

Figure 1. Assigning Score based on Terms

The rationale behind the scoring values is, again, that file names are treated as the most important elements and are assigned the highest score. When a BR term is identical to a file name, it is considered a full match (no further matches are sought for the file) and a relatively high score (adding 2) is assigned. The occurrence of the BR term in the file name is considered to be more important (0.025) than in the terms extracted from identifiers, method signatures or comments (0.0125).

The TT score values were chosen by applying the approach on a small sized training dataset, consisting of randomly selected 51 CRs from SWT and 50 CRs from AspectJ projects, i.e. (50+51)/4626=2.2% of all BRs, and making adjustments to the scores in order to tune the results for the training dataset.

4. EVALUATION RESULTS

We first ran ConCodeSe using just KP and TT scoring. Many BRs can be located by just assigning a high score to file names in certain positions of the BR summary, confirming the results of studies cited in the introduction that found file names mentioned in a large percentage of BRs [3, 8]. We then added ST scoring. The rank differences can be small but significant, moving a file from top-10 to top-5 or from top-5 to top-1. In some cases the file goes from not being in the top-10 to being the top-1, even if it is in the lowest scored (4^{th}) position in the ST.

1) Variation of the KP and ST Scores

We experimented further to evaluate the effects of our scoring mechanism by assigning different scores to the four key positions in the summary and in the ST with the following changes:

- The scores were halved.
- The scores were reversed.
- All 8 BR summary and ST positions have a uniform score.
- The scores were made closer to those of TT.

By keeping the values assigned at key positions relative to each other, e.g. by halving the scores, the results obtained were still better than those obtained by BugLocator and the other approaches. The reversed and uniform scoring led to the worst results. This confirms that the relative importance of the various positions (especially the first position) found through inspection of a few BRs of some projects applies to all projects. Using values close to the term scoring gives the best overall result out of the four variations for Eclipse, SWT and Tomcat, but still not as good as the original values used in Sections 3.1 and 3.2.

2) Overall Results

To answer our research question, we compare the performance of ConCodeSe against the state-of-the-art tools using their datasets and metrics (Table 3). Since only BugLocator is available, we couldn't obtain results for BLUiR and AmaLgam on the projects used by LOBSTER and LtR, and vice versa. LtR's top-N values for Tomcat were computed from the ranking results published in LtR's online annex [5]. LtR also used AspectJ, Eclipse and SWT but with a set of BRs and code files different from the BugLocator dataset. The LtR online annex includes all BRs but only Tomcat's code, so we were unable to run ConCodeSe on their versions of AspectJ, Eclipse and SWT.

Our tool outperforms BugLocator, BLUiR and LtR on all metrics for all projects, including BugLocator's MAP for ZXing, which BLUiR and AmaLgam weren't able to match. For LOBSTER, the authors report MAP and mean reciprocal rank (**MRR**) values obtained by varying the similarity distance in their approach, and we took their best values (0.17 and 0.25). LOBSTER is not a fully-blown bug localisation tool, it only investigates the added value of ST without using past history. To compare like for like, we ran our tool on ArgoUML using only the ST scoring (*ConCodeSe*ST). As Table 3 shows, even without using KP scoring ConCodeSe improves on their results. We believe the reason is that LOBSTER scores *all* files occurring in the stack trace, thus leading to a higher rank of the non-affected files,.

We also ran BugLocator on ArgoUML and Tomcat. In both cases ConCodeSe improved the results in all top-N categories. ConCodeSe also outperforms AmaLgam in all cases except for AspectJ's top-1 and MAP metrics. We note that ConCodeSe always improves the MRR value, which is an indication of how many files a developer has to go through in the ranked list before

finding one that needs to be changed. From the results we can answer the research question affirmatively: leveraging the occurrence of file names in BRs leads almost always to better performance than using project history and similar BRs.

Table 3. ConCodeSe compared to other tools

Project	Approach	Top-1	Top-5	Top-10	MAP	MRR
AspectJ	BugLocator	30.8%	51.0%	59.4%	0.22	0.41
	BLUiR	33.9%	52.4%	61.5%	0.25	0.43
	AmaLgam	44.4%	65.4%	73.1%	0.33	0.54
	ConCodeSe	**42.3%**	**68.2%**	**78.3%**	**0.33**	**0.67**
Eclipse	BugLocator	29.1%	53.8%	62.6%	0.30	0.41
	BLUiR	35.9%	56.2%	65.4%	0.33	0.44
	AmaLgam	34.5%	57.7%	67.0%	0.35	0.45
	ConCodeSe	**37.6%**	**61.2%**	**69.9%**	**0.37**	**0.57**
SWT	BugLocator	39.8%	67.3%	82.7%	0.45	0.53
	BLUiR	56.1%	76.5%	87.8%	0.58	0.66
	AmaLgam	62.2%	81.6%	89.8%	0.62	0.71
	ConCodeSe	**72.4%**	**89.8%**	**92.9%**	**0.68**	**0.94**
ZXing	BugLocator	40.0%	60.0%	70.0%	0.44	0.50
	BLUiR	40.0%	65.0%	70.0%	0.39	0.49
	AmaLgam	40.0%	65.0%	70.0%	0.41	0.51
	ConCodeSe	**55.0%**	**75.0%**	**80.0%**	**0.55**	**0.68**
Tomcat	BugLocator	42.1%	62.4%	71.0%	0.26	0.33
	LearnToRank	42.7%	62.6%	71.3%	0.49	0.55
	ConCodeSe	**51.5%**	**69.2%**	**75.4%**	**0.52**	**0.66**
ArgoUML	BugLocator	18.7%	42.9%	54.9%	0.11	0.40
	LOBSTER	n/a	n/a	n/a	0.17	0.25
	ConCodeSest	13.2%	48.4%	56.0%	0.20	0.33
	ConCodeSe	**31.9%**	**61.5%**	**65.9%**	**0.30**	**0.55**
ConCodeSe average		**48.5%**	**70.8%**	**77.1%**	**0.46**	**0.68**

5. DISCUSSION

Our approach makes 4 contributions to IR-based bug localisation.

First, contrary to other approaches, we don't treat the whole BR text uniformly. The name of the file being scored, if it occurs in the BR, gets a different treatment from other code file terms occurring in the BR. Moreover, certain positions of the BR summary of the ST, if exists, are also treated specially.

Second, while other approaches combine scores using a fixed weight for the whole application, we instead take always the best of several ranks for each file. In this way, we are not *a priori* fixing for each file whether the lexical scoring or the probabilistic VSM score should take precedence. We also make sure that stemming and comments are only taken into account for files where it matters. The use of the best of 8 scores is likely the reason for improving the key MRR metric across all projects.

Third, we leverage structure further than other approaches. Like BLUiR, we distinguish the BR's summary and description, but whereas BLUiR treats each BR field in exactly the same way (both are scored by Indri against the parts of a file) we treat each field differently, one with KP and the other with ST scoring.

Fourth, our approach addresses both the developer nature and the descriptive nature of BRs, which we observed in these projects and another project from the financial domain [1]. BRs of a developer nature tend to include technical details, like STs and file or method names, whereas BRs of a descriptive nature tend to rely on the user's domain vocabulary. By leveraging file names and STs when they occur in BRs, and by using the BR's terms when they don't, we cater for both types of BRs.

To sum up, whilst other localisation algorithms take a "one size fits all" approach, we treat each BR and file individually, using the summary, ST, stemming, comments and file names only when available *and* relevant, i.e. when they improve the ranking.

Threats to validity

We catered for *internal validity* by comparing the search performance of our tool like for like (i.e., using the same datasets and the same criteria) with five existing bug localisation tools [2, 3, 4, 5, 6]. Therefore, the improvement in results can only be due to our approach. It is conceivable that an IR engine using the LSI model may produce more or less sensitive results to using file names in BRs. We plan to experiment with LSI in future work.

We used a non-parametric Wilcoxon matched pairs statistical test to cater for *conclusion validity* since no assumptions were made about the distribution of the results. Based on the values obtained ($Z = -3.0955$, $W=0$ and $p=0.00194$), we conclude that on average ConCodeSe locates significantly ($p < 0.05$) more relevant source files in the top-*N*.

The characteristics of the projects (e.g. the domain, the identifier naming conventions, and the way comments and BRs are written, including the positions where file names occur) are a threat to *external validity*. We reduced this threat by repeating the search experiments with different applications, developed independently.

6. CONCLUDING REMARKS

This paper contributes a novel algorithm that, given a bug report (BR) and the application's source code files, uses a combination of lexical and structural information to suggest, in a ranked order, files that may have to be changed to implement the BR. We adopt others' ideas (separate summary from description [3], use stack traces [6]) and build on their findings (there is high occurrence of file names in BRs [3, 8]), but add our 2 key novel ideas (give higher score to file names and treat BRs individually by taking the best rank of several combinations of information) to obtain a more practical approach due to its improved results, its applicability *ab-initio*, without requiring history, and it offers simpler light-weight deployment compared to machine learning or tuning of weights.

We compared the results to five existing approaches, using the same BRs, applications and evaluation criteria, and found that overall our approach improved the ranking of the affected files, thereby increasing in a statistical significant way the percentage of BRs for which relevant files are placed among the top-1, 5, 10, which is 48.5%, 70.8% and 77.1% respectively, on average. We also improved, in most cases substantially, the mean reciprocal rank value for all six applications evaluated, thereby reducing the number of files to inspect before finding a relevant file.

Our IR-based approach can in principle also be applied to feature requests, because it doesn't depend on history, but the evaluation datasets only include bug reports. We plan to evaluate our approach with feature requests in future work.

7. ACKNOWLEDGMENTS

We thank H. Zhang for providing BugLocator and its datasets [2], R. Saha for the BLUiR dataset [3] and L. Moreno for the LOBSTER dataset [6]. This research is partially funded by ERC Advanced Grant 291652.

8. REFERENCES

[1] Dilshener, T., and Wermelinger, M., Relating developers' concepts and artefact vocabulary in a financial software module, In *Proc. 27th Int'l Conf. on Software Maintenance*, pp. 412-417, 2011.

[2] Zhou, J., Zhang, H., and Lo, D., 2012. Where Should the Bugs be Fixed?, In *Proc. 34th ICSE,* pp. 14-24, 2012.

[3] Saha, R., K., Lease, M., Khurshid, S. and Perry, D., E., Improving bug localization using structured information retrieval. In *ASE*, pp. 345–355, 2013.

[4] Wang, S., and Lo, D., Version history, similar report, and structure: Putting them together for improved bug localization, In *Proc. 22nd Int'l Conf. On Program Comprehension*, pp. 53-63, 2014.

[5] Ye, X., Bunescu, R., and Liu, C., Learning to rank relevant files for bug reports using domain knowledge, *Proc. 22nd Foundation of Software Engineering*, pp. 689–699, 2014.

[6] Moreno, L., Treadway, J., J., Marcus, A., and Shen, W., On the use of stack traces to improve text retrieval-based bug localization, In *Proc. 36th Int. Conf. Softw. Maint. Evol*., pp. 151–160, 2014.

[7] Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y. and Zhai, C., Have things changed now?: an empirical study of bug characteristics in modern open source software, In *Proc. 1st work. on ASS*, pp. 25-33, 2006.

[8] Schrötter, A., Bettenburg, N. and Premraj, R., Do stack traces help developers fix bugs?, In *Proc. 7th MSR'10*, pp. 118-121, 2010.

[9] Lucia, Thung, F., Lo,, D., and Jiang, L., Are faults localizable?, In *Proc. 9th MSR*, pp. 74–77, 2012.

[10] http://www.concodese.com

[11] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H., Exploring the influence of identifier names on code quality: an empirical study, *Proc. 14th European CSMR*, pp. 159–168, 2010.

[12] http://norm.al/2009/04/14/list-of-english-stop-words/

[13] http://lucene.apache.org/java/docs/index.html

[14] Rao, S. and Kak, A., Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. *Proc. 8th MSR*, p.43-52, 2011