10-2016

AmaLgam+: Composing rich information sources for accurate bug localization

Shaowei WANG

David LO
Singapore Management University, davidlo@smu.edu.sg

DOI: https://doi.org/10.1002/smr.1801

Citation

WANG, Shaowei and David LO. AmaLgam+: Composing rich information sources for accurate bug localization. (2016). *Journal of Software: Evolution and Process*. 28, (10), 921-942. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/3576

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2016; **28**:921–942
Published online 10 October 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1801
http://doi.org/10.1002/smr.1801

AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization

Shaowei Wang*,† and David Lo

School of Information Systems, Singapore Management University, Singapore

ABSTRACT

During the evolution of a software system, a large number of bug reports are submitted. Locating the source code files that need to be fixed to resolve the bugs is a challenging problem. Thus, there is a need for a technique that can automatically figure out these buggy files. A number of bug localization solutions that take in a bug report and output a ranked list of files sorted based on their likelihood to be buggy have been proposed in the literature. However, the accuracy of these tools still needs to be improved. In this paper, to address this need, we propose AmaLgam+, which is a method for locating relevant buggy files that puts together fives sources of information, namely, version history, similar reports, structure, stack traces, and reporter information. We perform a large-scale experiment on four open source projects, namely, AspectJ, Eclipse, SWT, and ZXing to localize more than 3000 bugs. We compare AmaLgam + with several state-of-the-art approaches including AmaLgam, BLUiR+, BRtracer+, BugLocator, and TFIDF-DHbPd. These approaches leverage one or several of the sources of information analyzed by AmaLgam+, but not all of them. On average, AmaLgam + achieves a 6.0% improvement over AmaLgam, which merges three sources of information, in terms of Mean Average Precision (MAP). For AspectJ and Eclipse datasets, in which there are many bug reports with stack traces and many reporters submit multiple bug reports, AmaLgam + achieves a 12.0% improvement over AmaLgam in terms of MAP. Compared with the other state-of-the-art approaches, AmaLgam + achieves an improvement of 20.3%, 22.5%, 33.1%, and 73.9% over BLUiR+, BRtracer+, BugLocator, and TFIDF-DHbPd in terms of MAP, respectively.

KEY WORDS: version history; similar report; structure; stack traces; reporter information; bug localization

1. INTRODUCTION

Software systems are often plagued with bugs. To improve the reliability of systems, developers often allow users to submit bug reports to bug tracking systems. Unfortunately, the number of these reports is often too large for the developers to handle manually in a timely manner. Anvik *et al.* cited a Mozilla triager that mentioned 'Everyday, almost 300 bugs appear that need triaging. This is far too much for Mozilla programmers to handle' [1]. One of the most time-consuming task to resolve a bug report is to find the buggy files that are responsible for a reported bug. A system may contain thousands or more files, and often only one or a few of these files need to be changed to fix a bug. Lucia *et al.* analyze 374 bugs from Rhino, AspectJ, and Lucene and find that 84–93% of the bugs reside in 1 and 2 source code files [2]. Thus, localizing these buggy files is like finding one or two needles in a big haystack.

To address the previous mentioned challenge, a number of studies have proposed ways to identify buggy program files given a bug report. Many of these approaches are information retrieval based, and

*Correspondence to: School of Information Systems, Singapore Management University, Singapore.
†E-mail: shaoweiwang.2010@smu.edu.sg

they work by computing similarities between a reported bug and source code files [3–8]. The source code files are then ranked based on their similarities to a reported bug.

Sisman and Kak leverage version history data for bug localization based on the intuition that files that are often buggy in the past are likely to be buggy in the future [5]. Several variants of their approach are proposed, and the best performing one is named TFIDF-DHbPd. Zhou *et al.* leverage similarities among bug reports for bug localization [8]. Given a new bug report, their approach, named BugLocator, finds files that are fixed to resolve similar older bug reports to locate buggy files of the new report. Saha *et al.* propose an approach named BLUiR, which leverages the structure of a bug report and a source code file [4]. BLUiR transforms a bug report and source code files to their constituent parts (i.e., a source code file is separated to four groups of identifiers, namely, class names, method names, variable names, and comments, and a bug report is separated to two groups of text, namely, summary and description) and employs structured information retrieval to compute their similarities. Saha *et al.* also propose BLUiR+ that extends BLUiR by leveraging similarities among bug reports following BugLocator. Also recently, Wong *et al.* present an approach named BRTracer, which separates stack trace from other text in a bug report and assigns a score to source code files by analyzing the stack trace [7]. Wong *et al.* also propose a BRTracer + that extends BRTracer by leveraging similarities among bug reports following BugLocator.

Despite the fact that many bug localization approaches are proposed in the literature, the accuracy of these approaches still needs to be improved. In this work, we would like to improve the accuracy of existing bug localization approaches by proposing a new approach named AmaLgam + that leverages five sources of information: version history, similar bug reports, structure, stack trace, and reporter information. The first four sources of information have been considered by prior works; however, none of them consider all of them together. Furthermore, we also add one additional source of information, namely, reporter information. Our hypothesis is a bug reporter that is likely to report issues affecting the same/similar software components. Thus, for new bug reports submitted by the same reporter, we make use of bug reports that the reporter has submitted before, to help identify packages that are more likely to contain buggy files. This paper substantially extends our preliminary work, named AmaLgam, which was published as an ICPC (International Conference on Program Comprehension) 2014 conference paper [6]. AmaLgam only considers three sources of information (i.e., version history, similar bug reports, and structure) while AmaLgam + considers five. We succinctly describe the similarities and differences between AmaLgam + and prior studies in Table I.

The way AmaLgam + combines historical information that is different from that of Sisman and Kak in the following respects:

1. Our approach uses a well-tested bug prediction formula that is used in Google, and it takes into consideration the effect of change burst [9].
2. Sisman and Kak consider the complete version history to compute a probability. Our approach only considers very recent version history and totally discards historical information that are more than k days away from the time a new bug report is submitted.
3. Sisman and Kak simply sums up the probability of a file to be buggy and the similarity of a bug report to the file. Our approach assigns weights that govern the contribution of the probability of a file to be buggy (computed by the bug prediction technique) and the similarity of a bug report to a file (computed by integrating BugLocator and BLUiR).

Table I. Comparison of our approach to state-of-the-art bug localization techniques.

Approach	Version history	Similar report	Structure	Stack trace	Reporter information
TFIDF-DHbPd [5]	Yes	No	No	No	No
BugLocator [8]	No	Yes	No	No	No
BLUiR [4]	No	No	Yes	No	No
BLUiR+ [4]	No	Yes	Yes	No	No
BRTracer [7]	No	No	No	Yes	No
BRTracer + [7]	No	Yes	No	Yes	No
AmaLgam [6]	Yes	Yes	Yes	No	No
AmaLgam+	Yes	Yes	Yes	Yes	Yes

Bug ID	76138
Open date	2004-10-12 21:53:00
Summary	**Ant editor** not following tab/space setting on shift right
Description	This is from 3.1 M2. I have **Ant**->**Editor**->Display tab width set to 2, insert spaces for tab when typing" checked. I also have **Ant**->**Editor**->Formatter->Tab size set to 2, and "Use tab character instead of spaces _unchecked_. Now when I open a build.xml and try to do some indentation, everything works fine according to the above settings, except when I highlight a block and press tab to indent it. It's the tab character instead of 2 spaces that's inserted in this case.
Fixed Files	org.eclipse.ant.internal.ui.editor.**AntEditor**.java org.eclipse.ant.internal.ui.editor.**AntEditorSourceViewerConfiguration**.java

Figure 1. An eclipse's bug report and the buggy source code files corresponding to it.

We have evaluated AmaLgam + on a dataset of more than 3000 bug reports from AspectJ, Eclipse, SWT, and ZXing. The AspectJ bug reports are taken from the iBugs benchmark [10], which have been used to evaluate the approaches of Sisman and Kak, Zhou *et al.*, Saha *et al.*, and Wong *et al*. The experiment results show that AmaLgam + can achieve an MAP (i.e., Mean Average Precision) scores of 0.40, 0.36, 0.62, and 0.41 for AspectJ, Eclipse, SWT, and ZXing bug reports, respectively. These results are better than those achieved by existing state-of-the-art approaches by 6.0% to 73.9%.

The contributions of our work are as follows:

1. We are the first to put together version history, similar reports, structure, stack trace, and reporter information for bug localization. Past bug localization studies have only used one or several (but not all) of these five sources of information.
2. We have evaluated our approach AmaLgam + on more than 3000 bug reports from four open source programs: AspectJ, Eclipse, SWT, and ZXing. Our experiments show that AmaLgam + can improve the MAP scores of existing state-of-the-art approaches by a substantial margin.

The structure of the remainder of the paper is as follows. In Section 2, we first present preliminary information on bug reports and some motivating examples. We elaborate the details of AmaLgam + in Section 3. We describe our experimental setup and results in Section 4. We describe related work in Section 5. We finally conclude and mention future work in Section 6.

2. PRELIMINARIES AND EXAMPLE

In this section, we first describe some preliminary information on bug reports. We then outline some text pre-processing steps that are applied to the bug reports. Finally, we show an example to illustrate why it is useful to consider version history, similar report, structure, stack trace, and reporter information.

2.1. Bug reports

A bug report is a document submitted by users to describe an error that they experience when they use a system. A bug report contains a number of fields; we are particularly interested in four of them, namely, bug identifier (id), the date a bug report was submitted (open date), summary of the error (summary), and more detailed description of the error (description).

We present a bug report from Eclipse in Figure 1, which can be downloaded from Eclipse's Bugzilla.[1] The identifier of this bug report is 76138, and it describes a problem with the ant editor,

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=76138

which does not follow a display setting. The bug ID provides a reference number that can be used to identify commits in version control systems that fix it, c.f. [8]. The open date helps us to identify bug reports that are submitted a number of days prior to bug 76138. The summary and description fields help us to understand the error that the user experienced.

]A bug localization tool takes as input a bug report and returns the potential buggy files. The corresponding buggy Java files for the bug report shown in Figure 1, which are identified by checking the corresponding bug fixing commits, are AntEditor.java and AntEditorSourceViewerConfiguration.java.

2.2. Text pre-processing

An information-retrieval based bug localization technique usually performs three pre-processing steps: text normalization, stopword removal, and stemming. The goal of the text pre-processing steps is to break a bug report or a source code file into terms that can then be analyzed by an information retrieval technique. In the pre-processing step, some compound words (e.g., program identifiers) are broken into parts, and some related words are mapped to the same term. We briefly describe these three pre-processing steps as follows.

First, text normalization would be performed which involves the removal of punctuation marks, tokenization (i.e., extraction of words from paragraphs or identifiers from source code), and identifier splitting. During this step, when a source code is processed, it would be converted into an Abstract Syntax Tree (AST), and using this tree, identifiers would be identified. These identifiers are split into its constituent words following Camel Case splitting [11]. For example, the identifier ‘getMethodName’ is split to ‘get’, ‘Method’, and ‘name’. In this study, both the split words and the full identifier name are kept. For example, for the class name ‘AntEditorSourceViewerConfiguration,’ which is one of the buggy files corresponding to the bug report shown in Figure 1, we convert it to six words: ‘ant’, ‘editor’, ‘source’, ‘viewer’, ‘configuration,’ and the full identifier name ‘AntEditorSourceViewerConfiguration’.

Second, we remove stopwords such as ‘on’,’the’, ‘are’, ‘is’, and so on. These stopwords carry little meaning and thus we remove them. Finally, we perform stemming which reduces inflected or derived words into a common root form. For example, the word “‘reading’ and ‘reads’ are reduced to the root form ‘read’. By doing this, similar words would be represented using the same term. We use the standard Porter Stemmer [12] to perform this stemming step.[2]

2.3. Motivating example

A traditional information retrieval (IR)-based bug localization approach usually first performs text pre-processing on a query (a bug report) and the documents in a corpus (source code files). Then, a similarity score between the query and each of the documents would be computed based on a particular information retrieval technique (e.g., TFIDF, latent Dirichlet allocation (LDA), and latent semantic indexing), for example, [3]. From Figure 1, we note that the buggy source code file names share a number of common words with the summary and description of the bug report, that is, ‘ant’ and ‘editor’. Based on these common words, a traditional IR-based bug localization approach would try to link the bug report with the source code files. In this sub-section, we highlight how version history, similar reports, structure, stack trace, and reporter information can be used to improve the accuracy of traditional IR-based bug localization techniques.

2.3.1. VERSION HISTORY

There are lots of historical data of changes to source code files that are stored in a version control system during program evolution. This historical data can be used to improve bug localization performance. Kim *et al.* found that bugs happen in bursts, and not in isolation [13]. The files responsible for a bug recently are more likely to be responsible for other bugs in the near future.

[2]http://tartarus.org/martin/PorterStemmer/

```
---------------------
hash:3532306
author:darins
commit_date:2004-10-12 04:28:35 +0000
message:Bug 76051 - Navigation to property resource or file

M    ant/org.eclipse.ant.ui/Ant
Editor/org/eclipse/ant/internal/ui/editor/AntEditor.java
---------------------
hash:3d1a68b
author:darins
commit_date:2004-10-07 01:02:22 +0000
message:Bug 50583 - Patternsets, path and fileset hovering (F2)

M    ant/org.eclipse.ant.ui/Ant
Editor/org/eclipse/ant/internal/ui/editor/AntEditorSourceViewerCon
figuration.java
A     ant/org.eclipse.ant.ui/Ant
Editor/org/eclipse/ant/internal/ui/editor/text/AntInformationProvider
.java
```

Figure 2. Recent commit logs prior to the reporting of bug report 76138.

Figure 2 presents the commit logs of Eclipse before bug 76138 occurred. We could see that the class files 'AntEditor.java' and 'AntEditorSourceViewerConfiguration.java', which were responsible for bug 76138, were also responsible for other bugs that happen prior to the reporting of bug 76138 (they are highlighted in bold). 'AntEditor.java' is fixed just 1 day prior to the reporting of bug 76138, and 'AntEditorSourceViewerConfiguration.java' is fixed just 7 days prior to the reporting of bug 76138. Thus, we could see that historical data can be used to better locate bug.

2.3.2. SIMILAR REPORTS

User often submits many similar bug reports that correspond to different errors that affect the same buggy program elements. For example, Figure 3 shows an older report with identifier 50303,[3] which were reported 9 months before bug report 76138. Note that this report shares the common words 'ant' and 'editor' with bug report 76138. Bug report 50303 was fixed on March 17, 2004 and was re-fixed on March 18, 2004, and 'AntEditor.java' was modified on both fix instances. By analyzing bug report 50303, we can obtain a hint on files that need to be changed to fix 76138. From the example, we could see that similar reports can be used to better locate bug.

2.3.3. STRUCTURE

Bug reports and source code files have structures. Bug reports have several fields including summary and description. Source code files can be split into class names, method names, variable names, and comments. This structural information can be leveraged for bug localization. Traditional IR-based bug localization approaches compute the similarity between a bug report and the entire content of a source code file (which contains a class name, many variable names, and many comments). For localizing the bug report in Figure 1, the class names contain the most important terms. Unfortunately, the impact of the terms 'ant' and 'editor' in the class names would be weaken by other tokens, which would make the performance poor. Structural information could be used to overcome this problem by computing the similarities of a query against different fields (e.g., class,

[3]https://bugs.eclipse.org/bugs/show $_{b}ug.\,cgi\,?\,id = 50303$

Bug ID	50303
Open date	2004-01-20 20:55
Summary	**Ant Editor** outline "Link with **Editor**"
Description	Similar to the Java **Editor** it would be a nice enhancement to have a "Link with **Editor**" toggle button for the **Ant Editor** outline page.
FixedFiles	org.eclipse.ant.internal.ui.editor.**AntEditor**.java 7 other files

Figure 3. An older eclipse's bug report and the buggy source code files corresponding to it.

method, variable, and comment) in a source code file separately and summing up those similarities. In this way, the tokens 'ant' and 'editor' would have stronger impact to the overall similarity. Thus, we could see that structure can be used to better locate bug.

2.3.4. STACK TRACE

A past study by Schroter *et al*. [14] shows that top 10 methods in stack traces are more likely to contain the root causes of bugs. Hence, stack trace information included in the bug report can be used as important information for bug localization. Figure 4 presents an example of a stack trace contained in a bug report of AspectJ. The buggy file for this bug report is 'AsmRelationshipProvider.java', which is located at the top of the stack trace. From this example, we can observe that we may be able to locate buggy files by analyzing the stack trace if it is included in a bug report.

2.3.5. REPORTER INFORMATION

In our dataset, 3479 bugs are reported by 2294 reporters. In other words, some reporters submitted more than one bugs. A user typically focuses on certain components or functionalities of a software system. Therefore, we could use information about buggy files of past bug reports that are submitted by a reporter to help to predict files, which are responsible for a new bug report that is submitted by the same reporter. Those buggy files are likely to be located close to one another in the directory/package structure (e.g., in the same package). In Table II, we present bug reports submitted by a reporter whose ID is `bpasero`. We can observe that buggy files of bug reports 100095 and 88717 are located within the same package, that is, 'eclipse.swt.dnd'.

3. APPROACH

In this section, we first describe the overall framework of AmaLgam+. We then present each of the five main components of AmaLgam+.

```
java.lang.NullPointerException
at org.aspectj.weaver.AsmRelationshipProvider.checkerMunger(AsmRelationshipProvider.java:51)
at org.aspectj.weaver.Checker.match(Checker.java:58)
at org.....BcelClassWeaver.match(BcelClassWeaver.java:985)
at org.aspectj.weaver.bcel.BcelClassWeaver.match(BcelClassWeaver.java:791)
at org.aspectj.weaver.bcel.BcelClassWeaver.weave(BcelClassWeaver.java:291)
at org.aspectj.weaver.bcel.BcelClassWeaver.weave(BcelClassWeaver.java:77)
at org.aspectj.weaver.bcel.BcelWeaver.weave(BcelWeaver.java:417)
at org.aspectj.weaver.bcel.BcelWeaver.weave(BcelWeaver.java:390)
...
...
at org.aspectj.ajdt.internal.core.builder.AjBuildManager.batchBuild (AjBuildManager.java:70)
at org.aspectj.ajde.internal.CompilerAdapter.compile(CompilerAdapter.java:103)
at org.aspectj.ajde.internal.AspectJBuildManagerCompilerThread.run(AspectJBuildManager.java:165)
```

Figure 4. Stack trace included in an Aspectj bug with ID 44117 – NPE on compile.

Table II. Buggy files of bug reports submitted by `bpasero`.

Bug ID	Fixed files
100095	eclipse.swt.dnd.URLTransfer.java
79481	eclipse.swt.custom.SashForm.java
88717	eclipse.swt.dnd.TableDragUnderEffect.java
	eclipse.swt.dnd.TreeDragUnderEffect.java

3.1. *Overall framework of AmaLgam+*

Figure 5 presents the overall framework of AmaLgam+. AmaLgam + takes as input a bug report to be localized (new bug report), a set of source code files of the system for which the bug report is submitted (source code files), a historyof commits made to the system as stored in a version control system (version history data), and a set of older bug reports stored in a bug tracking system (bug repository).

The inputs would be processed by five components of AmaLgam + ,namely: version history component (VHC), similar report component (SRC), structure component (SC), stack trace component (STC), and reporter information component (RIC). Version history component makes use of version history information to rank files. Similar report component makes use of older reports in bug repository to rank files. Structure component makes use of the structure of bug reports and source code files to rank files. Stack trace component makes use of the stack traces in the submitted bug report to rank files. Reporter information component makes use of reporter's previous bug reports to rank files. The five components each output a suspiciousness score for each source code file. These five sets of suspiciousness scores are input to the composer component, which produces the final ranked files.

3.2. *Version history component*

For the version history component, we make use of studies on bug prediction whose goal is to predict which files are likely to be buggy in the future, for example, [13, 15]. Kim *et al*. propose BugCache, which predicts future bugs by maintaining a relatively short list of most fault-prone program entities [13]. Rahman *et al*. propose a cheaper algorithm, which only sorts files based on the number of bug fixing commits that touch each of them [15]. Rahman *et al*. show that this simple and cheap

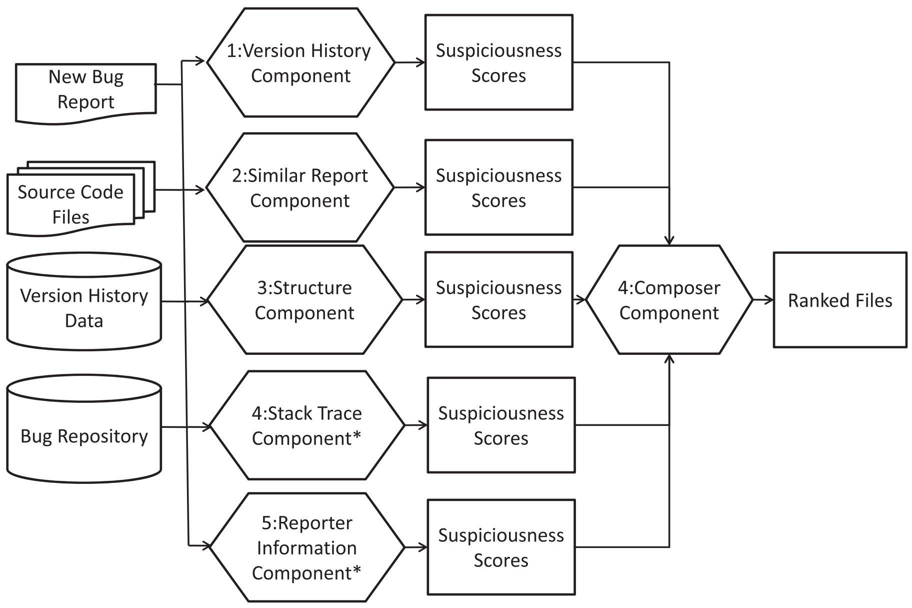


Figure 5. The overall framework of AmaLgam+. The asterisk symbol represents the newly added components over those proposed in our conference paper (i.e., AmaLgam) [6].

approach achieves almost the same performance as BugCache. Google developers adapt the simple algorithm proposed by Rahman *et al.* to predict bugs on their large systems [9]. The resulting algorithm is simple and fast. Thus, we decide to adapt this well-tested bug prediction algorithm of Google developers as our version history component. We briefly describe how we adapt this algorithm in the following paragraphs.

The algorithm takes as input commit logs and outputs a list of files with their suspiciousness scores. It first identifies relevant bug-fixing commits. The relevant bug fixing commits are identified by following two rules:

1. The commit log must match the following regular expression *regex*: $(.*fix.*)|(.*bug.*)$. This regular expression specifies that all commit logs containing the word 'fix' or 'bug' would be matched.
2. The commit must be made in the past k days.

We modify Google developers approach by including the second requirement. Our experience shows that including older bug-fixing commits do not affect performance much and even can slightly decrease performance. Also, it is computationally cheaper to onlyconsider recent commits. Parameter k could be set empirically. By default, we set k to 15. The algorithm analyzes these relevant commits and assigns a suspiciousness score to each source file f using the following equation:

$$score^H(f,k,R) = \sum_{c \in R \wedge f \in c} \frac{1}{1 + e^{12(1-((k-t_c)/k))}} \quad (1)$$

In the previous equation, R refers to the set of relevant commits, and t_c is the number of days that has elapsed between a commit c and the input bug report. The output of this algorithm is a set of suspiciousness scores, one for each file. We denote the suspiciousness score of file f assigned by the version history component as $Susp^H(f)$.

Example
Consider the input bug report in Figure 1 and the two commit logs in Figure 2. For simplicity sake, let us assume that there are no other commit logs. We would like to illustrate how Equation (1) is used to compute the suspiciousness scores of files AntEditor.java and AntEditorSourceViewerConfiguration.java. As both commits with identifiers 3532306 and 3d1a68b contain the word 'bug' and they are committed within 15 days before the time bug report 76138 was submitted (i.e., October 12, 2004, at 21:53:00), they are considered relevant bug fixing commits. The value of $(k - t_c)/k$ for commit 3532306 is 0.95 (because the commit was made around 17 h, that is, 0.7 day, before the time bug report 76138 was submitted). Thus, the suspiciousness score for AntEditor.java is 1.82. The suspiciousness score of AntEditorSourceViewerConfiguration.java can be computed in a similar way, and it is 0.009.

3.3. *Similar report component*

For our similar report component, we adapt BugLocator [8], in particular the algorithm that computes SimiRank scores. We describe briefly how we use this algorithm in the following paragraphs.

The algorithm takes in an input bug report and older bug reports that have been fixed in the bug repository. It then measures the similarity of the input bug report to the older fixed bug reports. Based on the similarity scores of the bug reports and the number of files that are modified to fix each bug report, we compute a suspiciousness score for each source code file.

To measure the similarity of two bug reports, the following steps are followed. First, each bug report is represented by their constituent pre-processed terms. Considering the universe of all terms as $\{t_1, \ldots, t_n\}$, we can compute for a bug report b, a vector $\rightarrow b$:

$$\rightarrow b = tf_b(t_1)idf(t_1), tf_b(t_2)idf(t_2), \ldots, tf_d(t_n)idf(t_n) \quad (2)$$

In the previous formula, $tf_b(t_i)$ corresponds to the number of times term t_i appears in bug report b, and $idf(t_i)$ corresponds to the reciprocal of the number of documents that contain term t_i. Given vector representations of two bug reports $\rightarrow b1$ and $\rightarrow b2$, their similarity can be measured by computing the standard cosine similarity [16] of their vector representations.

To compute a suspiciousness score for source code file *f*, we use the following equation:

$$score_R(f,b,B) = \sum_{b' \in \{b' | b' \in B \wedge f \in b'.Fix\}} \frac{sim(b,b')}{|b'.Fix|} \tag{3}$$

In the previous equation, *b* is the input bug report; *B* is the set of older fixed bug reports, and *sim*(*b*, *b*′) is the similarity of bug report *b* and *b*′, *b*′. *Fix* is the set of files that are modified to fix bug report *b*′, and |*b*′. *Fix*| is the size of set *b*′. *Fix*. The output of this algorithm is a set of suspiciousness scores, one for each file. In this component, we do not enforce a similarity threshold following what Zhou *et al*. did in their work [8]. We take all older bug reports to compute the suspiciousness score. We denote the suspiciousness score of file *f* assigned by the similar report component as $Susp^R(f)$.

Example
Consider the input bug report shown in Figure 1 and the older bug report in Figure 3. For simplicity sake, let us assume that there are no other bug reports in the bug repository. We would like to illustrate how Equation (3) is used to compute the suspiciousness scores of files AntEditor.java. Let us assume for simplicity sake that the similarity of the two bug reports is 0.15. The suspiciousness score of AntEditor.java can then be computed as 0.15/8 = 0.01875.

3.4. Structure component

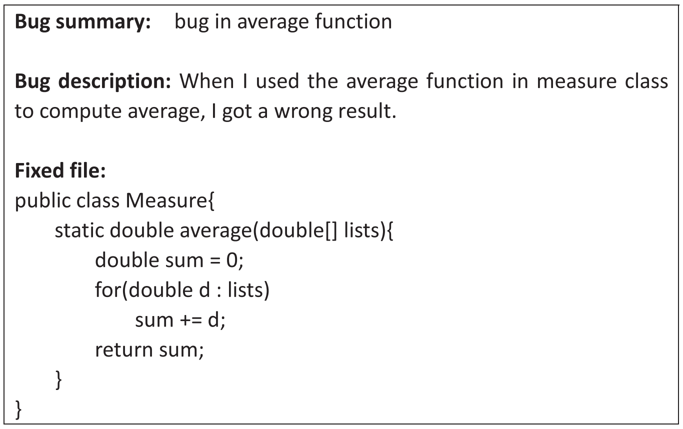
For the structure component, we use BLUiR [4] which performs structured retrieval for bug localization. For completeness-sake, we briefly describe BLUiR in the following paragraphs.

BLUiR breaks a bug report into two parts: summary and description. It breaks a source code file into four parts: class names, method names, variable names, and comments. Each of these parts can be converted into a vector following a similar procedure described in Section 3.3. The suspiciousness score of a source code file *f* given an input bug report *b* can then be computed as follows:

$$score_S(f,b) = \sum_{fp \in f} \sum_{bp \in b} sim(fp, bp) \tag{4}$$

where *fp* is a part of file *f*, *bp* is a field in bug report *b*, and *sim*(*fp*, *bp*) is the cosine similarity of the vector representations of *fp* and *bp*. The output of the structure component is a set of suspiciousness scores, one for each file. We denote the suspiciousness score of file *f* assigned by the structure component as $Susp^S(f)$.

Example
Consider a bug report and a file shown in Figure 6. After pre-processing (text normalization, stopword removal, and stemming), the terms in the summary field of the bug report are 'bug', 'averag', and


Bug summary: bug in average function

Bug description: When I used the average function in measure class to compute average, I got a wrong result.

Fixed file:
```
public class Measure{
    static double average(double[] lists){
        double sum = 0;
        for(double d : lists)
            sum += d;
        return sum;
    }
}
```

Figure 6. Example bug report and source code file.

'function'. The terms in the description field of the bug report are 'us', 'averag', 'function', 'measur', 'class', 'comput', 'got', 'wrong', and 'result'. The term in the class name of the file is 'measur'. The term in the method name of the file is 'averag'. The terms in the variable names of the file are 'list', 'sum', and 'd'. The set of terms in the comments of the file is Ø. Based on these fields of the bug report and these parts of the source code file, we can compute a suspiciousness score, which would be a summation of eight similarity scores.

3.5. Stack trace component

Bug reports may contain stack trace, which may provide clues for possible buggy files. Most IR-based bug localization approaches directly convert the entire bug description, which may include both text and stack trace, as a bag of words. Stack trace information is not separated from other text and analyzed differently. In this component, different from many existing approaches, we separate stack trace from other text and perform a specialized analysis on it. Our analysis is based on the intuition that the closer a reference to a file appears in a stack trace (i.e., the closer it is to the location of the failure/crash) the more likely the file is buggy.

To check for the presence of a stack trace in a bug report and to identify files that are referenced in the stack trace, we use a regular expression 'at [~ ()] + ([~ ()] + .java:[0–9]*)'. We are looking for the presence of a substring in the description of a bug report that starts with an 'at' and ending with a class name followed by a number indicating the line of error, which is located in between a pair of parentheses (e.g., at …Target.run(CflowCycles.java:24)). [~ ()] is used to match anything except a pair of parentheses. After the previous step, we obtain a set of java class files from a stack trace (if it exists). We then sort them based on their locations in the stack trace and remove the duplicate ones. After the above processing steps, we get an ordered set of Java files. Given the ranked files, the suspiciousness score of a source code file *f* in the stack trace of an input bug report *b* can then be computed as follows:

$$score_{ST}(f,b) = \begin{cases} 1/Rank_{f,b}, & if f\ is\ in\ the stack\ trace of\ b \\ 0 & , otherwise \end{cases} \tag{5}$$

In the previous equation, $Rank_{f,b}$ is the rank of file *f* in the stack trace contained in bug report *b*. The equation will assign a higher suspiciousness score to a file that appears closer to the top of the stack trace. We denote the suspiciousness score of file *f* assigned by the stack trace component as $Susp^{ST}(f)$.

Example

Consider the stack trace presented in Figure 4. The sorted set of Java files that appears in the stack trace along with their suspiciousness scores are shown in Table III.

3.6. Reporter information component

In this component, we make use of the reporter's information to compute to the suspiciousness of a given file. A user typically focuses on using one or a few functionalities of a software system rather than the whole range of functionalities provided by the system. This is especially true if the system

Table III. Sorted set of java files that appear in the stack trace shown in Figure 4 along with their suspiciousness scores.

Rank	File	Score
1	AsmRelationshipProvider	1.00
2	Checker	0.50
3	BcelWeaver	0.33
4	BcelClassWeaver	0.25
5	AjBuildManager	0.20
6	CompilerAdapter	0.17
7	AspectJBuildManager	0.14

is large and has quite a large number of functionalities. Based on this intuition, we compute the score of a file *f*, given a bug report *b*, submitted by reporter *r*, as follows:

1. Find past bug reports that are submitted by *r*.
2. Extract the names of the buggy files that are modified to fix the past bugs.
3. Extract the names of packages (denoted as $r.P$) that contain the buggy files.
4. Calculate the suspiciousness score of a file *f* following Equation (6).

$$score_{RI}(f, b, r) = \begin{cases} 1 & , iff \text{ is in } r.P \\ 0 & , otherwise \end{cases} \tag{6}$$

We denote the suspiciousness score of file *f* assigned by the reporter information component as $Susp^{RI}(f)$.

Example
To illustrate how the reporter information component works, consider the example shown in Table II. Let us consider bug report 100095, which is submitted by `bpasero`, as anew bug report. When this new bug report is processed, the reporter information component will first find past bug reports that are submitted by `bpasero` (i.e., 79481 and 88717). Next, this component will extract the buggy files that are modifiedto resolve bug reports 79481 and 88717. They are ‘eclipse.swt.custom.SashForm.java’, ‘eclipse.swt.dnd.TableDragUnderEffect.java’, and ‘eclipse.swt.dnd.TreeDragUnderEffect.java’. Then, we extract the package name of those buggy files, which are ‘eclipse.swt.dnd’ and ‘eclipse.swt.custom’. Finally, we assign score 1 to files in packages ‘eclipse.swt.dnd’ and ‘eclipse.swt.custom’ and score 0 to files in other packages.

3.7. Composer component

This component processes the five sets of suspiciousness scores output by the five components of AmaLgam + and computes a set of final suspiciousness scores. The composer component combines the scores output by the five components for a file *f* as follows:

$$\begin{aligned} &Susp^{S,R,H,ST,RI}(f) = \\ &w_1 \times Susp^S(f) + w_2 \times Susp^R(f) + w_3 \times Susp^H(f) + \\ &\qquad w_4 \times Susp^{ST}(f) + w_5 \times Susp^{RI}(f), \quad if \;\; Susp^S(f) \;\; or \;\; Susp^R(f) > 0 \\ &\qquad\qquad 0, \quad otherwise \end{aligned} \tag{7}$$

In the formula, we have five weights w_1, w_2, w_3, w_4, and w_5, which represent the contributions made by each component, to tune. In this work, we use genetic algorithm (GA) [17] to tune the weights by analyzing a training set of bug reports whose buggy files have been localized. GA works by first constructing an initial population of chromosomes; this population is then evolved by performing multiple iterations of selection, crossover, and mutation. In our setting, a chromosome is a combination of the five weights. In each iteration, a new population of chromosomes is created in which the ‘fittest’ chromosomes will be included. We create the initial population randomly (i.e., we randomly create chromosomes and added them to the initial population). We use the standard selection, crossover, and mutation operations described in [17]. The selection operation requires an objective function to guide the selection of the ‘fittest’ chromosomes. In effect, GA will try to search for a combination of five weights that maximize this objective function on the training data.

Before defining the objective function, we first define several commonly used metrics to measure the effectiveness of a bug localization approach as follows:

- Mean Average Precision: MAP is the most commonly used IR metric to evaluate ranking approaches. It considers the ranks of all buggy files into consideration. Therefore, MAP emphasizes all of the buggy files. MAP is computed by taking the mean of the *average precision* scores across all queries. The average precision of a single query is computed as follows:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{number\,of\,positive\,instances}, \tag{8}$$

where k is a rank in the returned ranked files, M is the number of ranked files, and *pos(k)* indicates whether the k^{th} file is a buggy file or not. *P(k)* is the precision at a given top k files and is computed as follows:

$$P(k) = \frac{\#buggy\,files}{k}. \tag{9}$$

- Mean Reciprocal Rank (MRR): The reciprocal rank for a query is the reciprocal of the position of the first buggy file in the returned ranked files. MRR is the mean of the reciprocal ranks over a set of queries Q, and it can be computed by following equation:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{10}$$

where $rank_i$ is the position of the first buggy file in the returned ranked files for the first query in Q.

Our goal is to maximize MAP and MRR on the training data; the higher their values are, the more effective a bug localization technique is. Thus, we define the objective function as follow:

$$ObjFunction = e^{(MAP+MRR)} \tag{11}$$

We give an equal weight to MAP and MRR because both of them are important. MAP is important if a developer is interested to find all buggy files by reading the recommended files one by one. MRR is important if a developer is interested in only finding the first buggy file.

In this work, we use a publicly available GA library called JGAP[4] to implement our approach. We set the values of maximum number of iterations and population size parameters to 200 and 50, respectively. We use JGAP default values for the other parameters.

In the end, the composer component would sort all source code files based on their final suspiciousness scores, and this ranked list of files would be the output of AmaLgam+.

4. EXPERIMENTS

In this section, we first describe the dataset that we use to evaluate our approach. Next, we describe our experimental settings, followed by our research questions. Finally, we describe our experiment results, which answer the research questions.

4.1. Dataset

We use the same dataset used by Wang *et al.*, Zhou *et al.*, and Saha *et al.* to evaluate BugLocator and BLUiR, respectively [4, 6, 8]. This dataset contains a total of 3379 bug reports from four popular open source projects, AspectJ, Eclipse, SWT, and ZXing. For each bug report, information on files that were modified to fix the bug is also provided in the dataset. The AspectJ bug reports originate from the iBugs benchmark [10] which was also used by Sisman and Kak to evaluate their proposed approach [5]. The AspectJ, Eclipse, and SWT bug reports were also used by Wong et al. to evaluate BRTracer+[7]. Table IV describes the dataset in more detail. For our version history component, we collect commit logs from Git repositories of those four projects.

4.2. Experimental setting

We compare the performance of AmaLgam+with a number of baselines: AmaLgam [6], BLUiR+ [4], BRTracer+[7], BugLocator [8], and TFIDF-DHbPd [5]. We use MAP and MRR presented in

[4]http://jgap.sourceforge.net/

Table IV. Dataset details.

Project	Description	Period	# of fixed bugs	# of source files	# of stack traces	# of reporter	# of terms in bug reports (Avg.)
AspectJ	Aspect-oriented extension of Java	07/2002–10/2010	286	6485	91 (32%)	101	233.1
Eclipse	Open source IDE	10/2004–03/2011	3075	12863	452 (15%)	2116	131.2
SWT	Open source widget toolkit	10/2004–04/2010	98	484	4 (4%)	61	112.6
ZXing	Barcode image processing library for Android platform	03/2010–09/2010	20	391	1 (5%)	16	218.5

Section 3.7 as two evaluation metrics. In addition, following prior bug localization studies [4, 6, 8], we also use Top-N Rank or Hit@N defined as follows:

1. Top-N Rank (Hit@N): This metric calculates the number of bug reports where one of the buggy files appears in the top N (i.e., 1, 5, and 10) ranked files. Given a bug report, if at least one of its buggy files is in the top N results, we consider the bug is successfully located. The higher the value of this metric is, the better the performance of an approach is.

In the experiment, we randomly sample 5% of the bug reports as the training data to train the weights w_1 to w_5 of AmaLgam+, and take the rest of the bug reports as the test data. Because of this random selection process and the fact that GA involves randomness, following the guidelines given by Arcuri and Briand [18], we repeat the experiment 100 times and report the average scores of the evaluation metrics. Note that the baselines do not require the tuning of weights and do not involve randomness and thus we do not repeat them 100 times and we use all of the bug reports to evaluate them. Although some of the baselines have weights, i.e., [4, 6, 8], these weights have been manually tuned and optimized on the dataset used to evaluate them (which is the same as our dataset). In effect, their parameters are manually tuned on all bug reports in the dataset, while ours are automatically tuned only on the training dataset.

We conduct all our experiments on a Windows 2008 server with 8 Intel R 2.53GHz cores and 24GB of RAM.

4.3. *Research questions*

Research Question 1 How effective is AmaLgam + for bug localization?

To answer this research question, we apply AmaLgam + to the four sets of bug reports in our dataset. We then evaluate the returned ranked lists and compute Hit@N, MAP, and MRR to characterize the effectiveness of AmaLgam + .

Research Question 2 Does AmaLgam + outperform other bug localization techniques?

In this research question, we compare the performance of AmaLgam + against five state-of-the-art approaches: AmaLgam by Wang and Lo [6], BLUiR+ by Saha *et al.* [4], BRtrace + by Wong *et al.* [7], BugLocator by Zhou *et al.* [8], and TFIDF-DHbPd by Sisman and Kak [5]. We compare the performance of AmaLgam + with the performance of these baselines reported in their original papers. We would like to investigate whether and to what extentAmaLgam + outperforms these existing state-of-the-art approaches.

Research Question 3 How complementary is each of AmaLgam + components?

In this research question, we want to analyze the complementarity of the five components of AmaLgam + by using Principal Component Analysis (PCA). PCA is a statistical technique able to identify various orthogonal dimensions (principal components) captured in the data (in our case: suspiciousness scores produced by each of AmaLgam + components) and the relative contributions of each of AmaLgam + components to each of these orthogonal dimensions. PCA has been used in many prior software engineering studies, for example, [19].

4.4. *Experiment results*

The following subsections describe our experimental results, which answer the five research questions. We answer one research question at a time.

4.4.1. RQ1: effectiveness of AmaLgam+. To answer RQ1, we measure the effectiveness of AmaLgam + in terms of the metrics we listed in Sections 3.7 and 4.2. Table V presents the results for all projects. For 141 (49.4%) AspectJ bug reports, AmaLgam + successfully locates a buggy source code file in the top 1 ranked file. For 208 (72.7%) AspectJ bugs, at least one buggy source code file is among the top 5 ranked files. For 230 (80.3%) AspectJ bugs, at least one buggy source code file is among the top 10 ranked files. In terms of MAP and MRR, AmaLgam + achieves a score of 0.40 and 0.60, respectively.

For Eclipse, 1097 (35.7%) and 1856 (60.3%) bugs could be localized by inspecting top 1 and 5 ranked files, respectively. Also, 2124 (69.1%) bugs could be localized when only the top 10 ranked files are inspected. The scores of MAP and MRR that AmaLgam + achieve for Eclipse are 0.36 and 0.47, respectively. For SWT, 61 (62.2%) bugs have a buggy file at the top 1 ranked file. Also, 79 (80.6%) and 88 (89.8%) bugs are successfully localized when only the top 5 and 10 ranked files are inspected, respectively. AmaLgam + achieves MAP and MRR scores of 0.62 and 0.71, respectively. For ZXing, AmaLgam + is able to localize 8 (40.0%), 13 (65.0%), and 14 (70%) bugs when only the top 1, 5, and 10 ranked files are inspected, respectively. In terms of MAP and MRR, AmaLgam + achieves a score of 0.41 and 0.51, respectively.

4.4.2. RQ2: AmaLgam versus other bug localization approaches. Table V compares the results of AmaLgam + with those of AmaLgam, BLUiR+, BRtrace+, BugLocator, and TFIDF-DHbPd in terms of Hit@1, Hit@5, Hit@10, MAP, and MRR.

Table V. Comparison among AmaLgam+, AmaLgam, BLUiR+, BRTracer+, BugLocator, and TFIDF-DHbPd. The most effective approaches for the four project are highlighted in bold font.

Project	Approach	Hit@1	Hit@5	Hit@10	MAP	MRR
AspectJ	AmaLgam+	**141 (49.4%)**	**208 (72.7%)**	**230 (80.3%)**	**0.40**	**0.60**
	AmaLgam	127 (44.4%)	187 (65.4%)	209 (73.1%)	0.33	0.54
	BLUiR+	97 (33.9%)	150 (52.4%)	176 (61.5%)	0.25	0.43
	BRTracer+	113 (39.5%)	173 (60.5%)	197 (68.9%)	0.29	0.49
	BugLocator	88 (30.8%)	146 (50.1%)	170 (59.4%)	0.22	0.41
	TFIDF-DHbPd	N/A	N/A	N/A	0.23	N/A
Eclipse	AmaLgam+	**1097 (35.7%)**	**1856 (60.3%)**	**2124 (69.1%)**	**0.36**	**0.47**
	AmaLgam	1060 (34.5%)	1775 (57.7%)	2059 (67.0%)	0.35	0.45
	BLUiR+	1013 (32.9%)	1729 (56.2%)	2010 (65.4%)	0.33	0.44
	BRTracer+	1002 (32.6%)	1719 (55.9%)	2005 (65.2%)	0.33	0.43
	BugLocator	896 (29.1%)	1653 (53.8%)	1925 (62.6%)	0.30	0.41
SWT	AmaLgam+	**62 (63.3%)**	79 (80.6%)	**88 (89.8%)**	**0.62**	**0.71**
	AmaLgam	61 (62.2%)	**80 (81.6%)**	**88 (89.8%)**	**0.62**	**0.71**
	BLUiR+	55 (56.1%)	75 (76.5%)	86 (87.8%)	0.58	0.66
	BRTracer+	46 (46.9%)	78 (79.6%)	87 (88.8%)	0.53	0.60
	BugLocator	39 (39.8%)	66 (67.3%)	81 (62.6%)	0.45	0.53
ZXing	AmaLgam+	**8 (40.0%)**	**13 (65.0%)**	**14 (70.0%)**	0.41	**0.51**
	AmaLgam	**8 (40.0%)**	**13 (65.0%)**	**14 (70.0%)**	0.41	**0.51**
	BLUiR+	**8 (40.0%)**	**13 (65.0%)**	**14 (70.0%)**	0.39	0.49
	BRTracer+	NA	NA	NA	NA	NA
	BugLocator	**8 (40.0%)**	12 (60.0%)	**14 (70.0%)**	**0.44**	0.50

4.2.2.1. AMALGAM + VERSUS AMALGAM

For AspectJ, in terms of Hit@1, AmaLgam + achieves a 11.0% improvement over AmaLgam score. By only inspecting the top 5 and 10 ranked files, using AmaLgam+, a debugger can locate 21 more bugs than if he/she uses AmaLgam; this corresponds to an improvement of Hit@5 and Hit@10 of 11.2% and 10.0%, respectively. In terms of MAP and MRR, AmaLgam + improves AmaLgam's scores by 21.2% and 11.1%, respectively. For Eclipse, AmaLgam + improves AmaLgam by 3.5%, 4.6%, 3.2%, 2.9%, and 4.4% in terms of Hit@1, Hit@5, Hit@10, MAP, and MRR, respectively. For SWT and ZXing bug reports, AmaLgam + and AmaLgam have the same/similar effectiveness in terms of each of the metrics.

On average, AmaLgam + achieves a 6.0% and 3.9% improvement over AmaLgam in terms of MAP and MRR, respectively. If we only consider AspectJ and Eclipse, on average, AmaLgam + achieves a 12.0% and 7.8% improvement over AmaLgam in terms of MAP and MRR, respectively. From these results, we can see that AmaLgam + is able improve AmaLgam when there is enough information coming from stack traces and reporters.

4.2.2.2. AmaLgam + versus other baselines. Comparing AmaLgam + with BLUiR+, we could note that AmaLgam + consistently outperforms BLUiR+ in terms of MAP and MRR for all programs. The Hit@N scores of AmaLgam + is better than those of BLUiR+ for all programs except ZXing. For ZXing, the Hit@N scores of AmaLgam + are the same with those of BLUiR+. On average, AmaLgam + improves the MAP and MRR scores of BLUiR+ by 20.3% and 14.5%, respectively. We can note that AmaLgam + consistently outperforms BRTracer + in terms of Hit@N, MAP, and MRR for three programs (we exclude ZXing, as BRTracer + was not evaluated on ZXing bug reports). On average, AmaLgam + improves the MAP and MRR scores of BRTracer + by 21.3% and 16.7%, respectively.

Comparing our approach with BugLocator, AmaLgam + outperforms BugLocator with respect to all metrics for AspectJ, Eclipse, and SWT bug reports. Both techniques have the same performance in terms of Hit@1, and Hit@10 for ZXing. For ZXing, AmaLgam + improves BugLocator in terms of MRR and Hit@5 but marginally loses to BugLocator in terms of MAP. On average, AmaLgam + improves the MAP and MRR scores of BugLocator by 33.1% and 24.2%, respectively. For TFIDF-DHbPd, Sisman and Kak only evaluates it using AspectJ bug reports from the iBugs benchmark. They also did not compute Hit@N or MRR. Because TFIDF-DHbPd code is not publicly available and Sisman and Kak evaluate the performance of TFIDF-DHbPd only on AspectJ, in the table, we only show the MAP score of TFIDF-DHbPd for AspectJ. Comparing our approach with TFIDF-DHbPd, we can improve their approach's MAP score on AspectJ bug reports by 73.9%.

4.2.2.3. Statistical tests. We have performed Wilcoxon signed-rank test [20] to test whether the improvement obtained by AmaLgam + over AmaLgam (the best performing baseline) is significant with p-value set to 0.5. We find that the improvements in terms of MAP and MRR are significant (p-value < 0.0001). To investigate if the differences of MAP and MRR scores of AmaLgam + and AmaLgam are substantial, we also compute Cohen's d [21], which measures effect size. Cohen defined an effect size of 0.2, 0.5, and 0.8 to be small, medium, and large, respectively. If the effect size is close to 0, it means that the difference is not substantial. We find that the effect sizes for the MAP and MRR differences are 0.20 and 0.21 respectively, which indicate small but substantial differences. If we only consider AspectJ and Eclipse bug reports, where there are many with stack traces and many bug reporters submit more than one bug report, the effect sizes for the MAP and MRR differences are 2.05 and 0.66, respectively, which indicate large and medium differences, respectively.

4.2.2.4. Further analysis. Comparing AmaLgam + and AmaLgam, the results for SWT are not as encouraging as the results for AspectJ and Eclipse, because only four SWT bug reports contain stack traces and only 17 reporters submit more than one bug report. Similarly the results are not as encouraging for ZXing, because there is only one bug report that contains a stack trace, and only one reporter submits more than one bug report. Thus, for SWT and ZXing, the stack trace and reporter information components contribute very little to the final results.

Furthermore, we note that for ZXing, the results for all baselines (AmaLgam, BLUiR+, and BugLocator) and AmaLgam + are very similar. Upon manual inspection, we find that the suspiciousness scores generated by three components of AmaLgam + (i.e., version history, stack trace, and reporter information components) are close to zeroes or are relatively small to affect the overall suspiciousness scores. Thus, version history, stack trace, and reporter information do not help much. We have described the reasons why the stack trace and reporter information components do not work well in the previous paragraph. The version history component does not work well because few commits were made in the 15 days prior to the submission of most of ZXing bug reports. Because the performance of BLUiR+ and BugLocator is similar, we can also infer that the effectiveness boost gained by performing structured information retrieval is also not much. This is because most of the relevant class names and method names (or their constituent words) do not appear in the summary or description of ZXing bug reports. BLUiR+ works by splitting identifier names into groups such that less important groups (e.g., comments, variable names) do not create noise that affects the more important ones (e.g., class names, method names). Unfortunately, for ZXing, identifiers in the more important groups do not match words that appear in the summary and description fields. We have also contrasted the performance of BLUiR and BLUiR+ (not shown in Table V) and find that their performance also does not differ much. This suggests that the benefit gained by leveraging similar bug reports is not much. This is the case because compared with Eclipse, AspectJ, and SWT, there are much fewer bug reports for ZXing; thus, only very few similar bug reports could be found when a new bug report is submitted. The results for ZXing need to be considered with a grain of salt although because there are only 20 bug reports in the ZXing dataset.

In most cases, AmaLgam + performs better than the other baselines, which demonstrate that our approach could better locate buggy files. For example, for the AspectJ bug report with ID 44117 whose stack trace is shown in Figure 4, AmaLgam + can locate the buggy file org.aspectj.weaver. AsmRelationshipProvider.java at the top 1 position in the result list because it appears in the stack trace. However, many other baselines cannot locate it. For example, for the result list produced by BLUiR+, the buggy file does not appear in the first 50 files. However, in a few cases, some components in AmaLgam + can bring in noise to the final results and make it performs worse than other baseline approaches. For example, the version history component can bring in some noise to adversely affect the final results. When locating the buggy files for the AspectJ bug report with ID 36234, BLUiR+ ranks the buggy file org.aspectj.tools.ajc.Main.java at the fourth position. However, AmaLgam + ranks it at the sixth position. This is because the version history component assigns two files (i.e., AjBuildConfig.java and AjState.java from org.aspectj.ajdt.internal.core.builder package) with scores higher than that of the buggy file as they were frequently modified in the previous 15 days before bug report 36234 was submitted.

4.4.3. RQ3: complementarity analysis on the components of AmaLgam+. We use PCA to analyze the complementarity of five components (i.e., VHC, SRC, SC, STC, and RIC) of AmaLgam+. Table VI presents the results obtained by comparing the suspiciousness scores of each AmaLgam + component on the four datasets. We can note that PCA identifies five principal components.

The first principal component (C1) explains (i.e., accounts for) the majority of the overall variability ranging from 66.21% to 81.45% (across the four datasets). We can notice that the SC contributes most to C1. The second principal component (C2) explains 14.69% to 23.95% of the overall variability. Among the four datasets, we can note that the VHC contributes most to C2 for AspectJ and SWT datasets, while SRC contributes most to C2 for Eclipse and ZXing datasets. The first two components explain at least 85.98% of the overall variability across the four datasets. For C3, the STC contributes the most for AspectJ and ZXing datasets, while SRC and VHC contribute more than other components for SWT and Eclipse datasets, respectively. For C4, different components contribute the most for different datasets. For C5, only the RIC contributes to it across the four datasets.

The results show that the structure component makes the most contributions to capture the variability in the datasets. The reporter information component makes the least contribution. The other three components perform differently on different datasets. Additionally, because the

Table VI. Results of the principal component analysis.

	AspectJ					Eclipse					**SWT**					**ZXing**				
	C1	C2	C3	C4	C5	C1	C2	C3	C4	C5	C1	C2	C3	C4	C5	C1	C2	C3	C4	C5
VHC	−0.010	**0.139**	−0.006	−0.004	0.000	−0.002	0.004	**0.063**	−0.006	0.000	0.007	**0.120**	−0.002	−0.001	0.000	0.006	−0.005	−0.006	**0.031**	0.000
SRC	0.008	0.011	0.042	**0.066**	0.000	−0.002	**0.171**	−0.001	−0.001	0.000	0.018	0.004	**0.068**	0.000	0.000	0.026	**0.058**	0.002	0.002	0.000
SC	**0.315**	0.004	−0.002	−0.002	0.000	**0.405**	0.001	0.000	0.000	0.000	**0.199**	−0.004	−0.006	0.000	0.000	**0.111**	−0.013	−0.002	−0.003	0.000
STC	0.001	0.005	**0.075**	−0.037	0.000	-0.002	0.003	0.006	**0.061**	0.000	−0.001	0.002	0.001	**0.034**	0.000	0.006	−0.004	**0.036**	0.005	0.000
RIC	0.000	0.000	0.000	0.000	**0.011**	0.000	0.000	0.000	0.000	**0.007**	0.000	0.000	0.000	0.000	**0.008**	0.000	0.000	0.000	0.000	**0.020**
VE	75.22%	14.69%	5.66%	4.33%	0.089%	81.45%	14.63%	2.0%	1.90%	0.006%	67.61%	18.37%	6.83%	5.15%	2.03%	66.21%	23.95%	7.78%	1.95%	0.007%

Values in bold show AmaLgam + components that best capture the main principal components (orthogonal dimensions) identified by PCA.
C1 to C5, principal components 1 to 5; VHC, version history component; SRC, similar report component; SC, structure component; STC, stack trace component; RIC, reporter information component; VE, variability explained. The maximum value of each column is highlighted in bold font.

contributions of each PCA component across the four datasets are non-zeroes and each of the AmaLgam + components contributes to at least one PCA component, none of the AmaLgam + components are useless in explaining the variability in the datasets. Furthermore, we can note that for each PCA component, one of the AmaLgam + components significantly dominates the rest (i.e., its contribution is significantly larger than other AmaLgam + components); this shows that the components are reasonably orthogonal to one another and are not rendered redundant due to the presence of other components.

4.5. *Threats to validity*

Threats to internal validity include experimenter bias. To reduce this threat, we reuse the bug report dataset that has been used before to evaluate prior approaches. Thus, the evaluation is not biased to our approach.

Threats to external validity relate to the generalizability of our findings. To reduce this threat, we have analyzed more than 3000 bug reports from four popular projects. Still in the future, we plan to reduce these threats further by analyzing more bug reports from more projects written in multiple programming languages.

Threats to construct validity refer to the suitability of the set of evaluation metrics that we use in this study. Three metrics are used namely Hit@N, MAP, and MRR. These metrics are well-known information retrieval metrics and have been used before to evaluate many past bug localization approaches, for example, [3–5, 8]. Thus, we believe there is little threat to construct validity.

5. RELATED WORK

In this section, we first describe a number of bug localization works. We then describe some bug prediction and feature location works. The survey here is by no means complete.

5.1. *Bug localization*

In recent years, many bug localization approaches have been proposed. These methods can be categorized into two: dynamic and static approaches.

Generally, dynamic approaches can localize a bug much more precisely than static approaches, for example, pinpoint a buggy statement or basic block. However, they usually require a test suite to execute a program to collect passing and failure execution traces. Thus, the effectiveness of a dynamic approach is often dependent on the quality of a test suite. Unfortunately, Kocchar *et al.* have shown that the adoption of software testing in many projects is often poor [22]. Spectrum-based fault localization, for example, [23–26] and model-based fault localization, for example, [27, 28], are some of the well-known dynamic approaches. Spectrum-based fault localization approaches often use program traces to correlate program elements at various granularity levels (e.g., statements, basic blocks, functions, and components) with program failures often with the help of a statistical analysis. Tarantula [24] and Ochiai [1] are two well-known techniques, and they are proposed to rank program elements according to their suspiciousness scores computed based on the executions of a program with a test suite. The basic idea of Tarantula and Ochiai is that a program element is considered to be more suspicious if it appears more frequently in failed executions than in correct ones. Saha *et al.* propose a customized automated fault localization technique for data-centric programs, which interact with databases [26]. Model-based fault localization approaches, for example, [27, 28] are based on more expensive logic reasoning over formal models of programs, which are often more accurate than spectrum-based fault localization approaches.

Static approaches do not require any test suite to be run to generate execution traces. They only need program source code files and bug reports to localize a bug. The static approaches usually can be categorized into two groups: program analysis-based approaches and IR-based approaches. FindBugs is a program analysis-based approach that locates a bug based on some predefined bug patterns [29]. However, it often detects too many false positives and misses many real bugs [30]. IR-based approaches use information retrieval techniques (such as TFIDF, LSA, and LDA) to calculate the similarity between a bug report and a source code file. Rao and Kak investigate many standard information retrieval techniques for bug localization and find that simpler techniques, for example, TFIDF and SUM, perform the best [3]. Lukins *et al.* use LDA, which is a well-known topic modeling approach, to localize bug [31]. Wang *et al.* propose to compose various vector space models with various term frequency-inverse document frequency (tf-idf) weighting schemes where the weights of the models are learned using a genetic algorithm (GA) [32]. They demonstrate that the compositional vector space model can outperform standard vector space model when it is being used individually or integrated with another bug localization technique.

Sisman and Kak propose a history-aware IR-based bug localization solution to achieve a better result [5]. Zhou *et al.* propose BugLocator, which leverages similarities among bug reports and uses a refined vector space model to perform bug localization [8]. Saha *et al.* consider the structure of bug reports and source code files and employ structured retrieval to achieve a better result [4]. Moreno *et al.* present an approach, namely, Lobster, which uses a text retrieval based technique and stack trace analysis to perform bug localization [33]. To locate buggy files, Lobster combines the textual similarity between a bug report and a code unit and the structural similarity between the stack trace and the code unit. The structural similarity is measured as the shortest path from the elements in the stack trace and the target code unit. Wong *et al.* propose an approach that uses a variant of vector space model and stack trace analysis to perform bug localization [7]. They extract the stack traces and compute a suspiciousness boost score of each file in the stack traces. For files that are listed among the top-10 files in the stack trace, the suspiciousness boost scores of the files are the reciprocal of their ranks; on the other hand, the suspiciousness scores of other files are set to either 0.1 (if they also appear in the stack trace or are directly used by a file in the stack trace) or 0. Different from the existing IR-based bug localization approaches, we put together version history, similar report, structure, stack traces, and reporter information to achieve better performance.

5.2. Bug prediction

There are many approaches proposed for bug prediction. One family of bug prediction approaches uses change logs to predict buggy files. Change log-based approaches extract historical information from a version control system and assume that recently or frequently changed files have the most potential to be buggy. Hassan measures the complexity of a code change and proposes several code change models, which are based on the concept of entropy and show that the code change models can be used to predict future faults [34]. Kim *et al.* propose BugCache, which stores a list of recent buggy files in a cache and uses it to predict future buggy files [13]. BugCache is based on an assumption that similar bugs happen in bursts and not in isolation. Rahman *et al.* perform an empirical study to evaluate BugCache and show that it is not substantially better than a basic prediction model, which computes the suspiciousness of a file based on the number of bug-fixing commits that touch the file [15].

Another family of approaches does not require historical data but only analyzes the current version of a system using various metrics. One well-known set of metrics is the Chidamber and Kemerer metrics [35]. These metrics and several coupling metrics have been used by El Emam *et al.* to predict faults on commercial Java application [36]. Nagappan *et al.* use a number of source code metrics (including Chidamber and Kemerer metrics) to predict module-level defects on five Microsoft systems [37]. They find that no predictor could perform well on all the projects. Marcus *et al.* propose the notion of conceptual cohesion of classes (C3), which is based on the analysis of unstructured text (e.g., comments and identifiers) in a code base, and use C3 for defect prediction [38].

There are also other approaches that do not belong to the two families described earlier. For example, Zimmermann and Nagappan use network analysis to analyze the dependencies between binaries in Windows server 2003 and predict defects based on that analysis [39].

5.3. Feature/concept/concern location

Feature/concept/concern location is a task that is closely related to bug localization. Its goal is to map a description of a feature or concept or concern to the program units (e.g., package, file, and method) that implement it. Many approaches have been proposed to perform feature/concept/concern localization with information retrieval techniques.

Poshyvanyk *et al.* make use of latent semantic indexing to map a software feature to its relevant program units and then apply Formal Concept Analysis to cluster the results [40]. In another work, they also make use of execution traces in addition to textual description of a feature to locate relevant program units [41]. Dit *et al.* combine information retrieval, execution, and link analysis algorithms to improve feature location techniques that analyze textual description and execution traces by using data fusion model [42]. Gethers *et al.* combine IR, dynamic analysis, and software repository mining techniques to recommend relevant source code entities given a change request and its contextual information, that is, execution trace and initial source code entity to be changed [43]. Wang *et al.* perform an empirical study on Linux kernel to evaluate the performance of 10 different IR models for feature location [44]. They show that vector space model outperforms other models. There are many other feature location approaches. For a comprehensive description of these studies, please refer to a recent survey paper by Dit *et al*. [45].

6. CONCLUSION AND FUTURE WORK

A large number of bug reports are submitted during the evolution of a software system. For a large system, locating the source code files responsible for a bug is a tedious and expensive work. Thus, there is a need to develop a technique that can automatically figure out these buggy files given a bug report. A number of bug localization tools have been proposed in recent years. However, the effectiveness of these tools still needs to be improved further. In this paper, to localize bugs more effectively, we propose a new approach named AmaLgam + which integrates five sources of information: version history, similar reports, structure, stack trace, and reporter information.

We perform a large-scale experiment on more than 3000 bugs in four projects, namely, AspectJ, Eclipse, SWT, and ZXing, to evaluate the effectiveness of AmaLgam + and compare it with a number of existing state-of-the-art bug localization approaches. Compared with a history-aware bug localization approach proposed by Sisman and Kak, AmaLgam + achieves a 73.9% improvement in terms of MAP. Compared with BugLocator, which considers similar reports, AmaLgam+, on average, achieves a 33.1% improvement in terms of MAP. Compared with BLUiR+, which considers structural information and similar reports, AmaLgam+, on average, achieves a 20.3% improvement in terms of MAP. Furthermore, AmaLgam + also outperforms BRTracer+, which considers stack trace information and similar reports, on average by 21.3% in terms of MAP. AmaLgam + boosts the performance of the preliminary version of this work, named AmaLgam [6], which integrates three sources of information (i.e., version history, similar reports, and structure); for AspectJ and Eclipse bug reports, on average, AmaLgam + outperforms AmaLgam by 12.0% in terms of MAP.

In the future, we would like to reduce the threats to external validity further by applying our approach on more bug reports from various systems. We plan to do a large scale empirical study using data from at least 20 projects and investigate the effectiveness of 5–10 existing bug localization approaches. We are also interested to integrate other bug localization approaches to AmaLgam + .

ACKNOWLEDGEMENTS

We would like to thank Zhou *et al.* and Saha *et al*. for making the detailed results of BugLocator and BLUiR + and datasets used to evaluate BugLocator and BLUiR+ available for public download. We also would like to thank the reviewers of ICPC 2014 and Journal of Software: Evolution and Process for their valuable comments and advice.

REFERENCES

1. Anvik J, Hiew L, Murphy GC. Coping with an open bug repository. *ETX* 2005; 35–39.
2. Lucia, Thung F, Lo D, Jiang L. Are faults localizable? In MSR, 2012; 74–77.
3. Rao S, Kak AC. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In MSR, 2011; **43–52**.

4. Saha RK, Lease M, Khurshid S, Perry DE. Improving bug localization using structured information retrieval. In ASE, 2013; 345–355.
5. Sisman B, Kak AC. Incorporating version histories in information retrieval based bug localization. In MSR, 2012; 50–59.
6. Wang S, Lo D. Version history, similar report, and structure: putting them together for improved bug localization. In 22nd International Conference on Program Comprehension, ICPC 2014*, Hyderabad, India, June 2-3*, *2014*, 53–63, 2014.
7. Wong C, Xiong Y, Zhang H, Hao D, Zhang L, Mei H. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), pages 181–190, October 2014.
8. Zhou J, Zhang H, Lo D. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In ICSE, **14–24**, 2012.
9. Lewis C, Ou R. Bug prediction at Google. http://google-engtools.blogspot.sg/2011/12/bug-prediction-at-google.html, 2011.
10. Dallmeier V, Zimmermann T. Extraction of bug localization benchmarks from history. *ASE* 2007; 433–436.
11. Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 2002; **28**(11):970–983.
12. M Porter. An algorithm for suffix stripping. *Program*, **14**(3):130–137, 1997.
13. Kim S, Zimmermann T, Whitehead Jr. EJ, Zeller A. Predicting faults from cached history. In Proceedings of the 29th International Conference on Software Engineering, ICSE ′07,2007; 489–498.
14. SchrÃ¶ter A, Bettenburg N, Premraj R. Do stack traces help developers fix bugs? In J. Whitehead and T. Zimmermann, editors, MSR, 2010; 118–121.
15. Rahman F, Posnett D, Hindle A, Barr E, Devanbu P. Bugcache for inspections: Hit or miss? In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE ′11, 322–331, New York, NY, USA, 2011. ACM.
16. Baeza-Yates RA, Ribeiro-Neto B. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co.: Inc., Boston, MA, USA, 1999.
17. Holland JH. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press: Cambridge, MA, USA, 1992.
18. Arcuri A, Briand LC. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 2014; 219–250.
19. Bavota G, Lucia AD, Marcus A, Oliveto R. Using structural and semantic measures to improve software modularization. *Empirical Software Engineering* 2013; 901–932.
20. Wilcoxon F. Individual comparisons by ranking methods. *Biometrics Bulletin* 1945; **6**(1:80â€"83):.
21. Cohen J. *Statistical Power Analysis for the Behavioral Sciences*, 2nd edn. Routledge, 1988.
22. Kochhar PS, Bissyandé TF, Lo D, Jiang L. An empirical study of adoption of software testing in open source projects. In QSIC, 2013; 103–112.
23. R Abreu, P Zoeteweij, R Golsteijn, and AJC van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, **82**(11):1780–1792, 2009.
24. Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE ′05, 2005.
25. Lucia, Lo D, Jiang L, Budi A. Comprehensive evaluation of association measures for fault localization. In ICSM, 2010; 1–10.
26. Saha D, Nanda MG, Dhoolia P, Nandivada VK, Sinha V, Chandra S. Fault localization for data-centric programs. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE ′11, 2011.
27. Feldman A, van Gemund A. A two-step hierarchical algorithm for model-based diagnosis. In Proceedings of the 21st National Conference on Artificial Intelligence – Volume 1, AAAI′06, 2006; 827–833.
28. Mayer W, Stumptner M. Model-based debugging – state of the art and future challenges. *Electronic Notes in Theoretical Computer Science* 2007; **174**(4):61–82.
29. D Hovemeyer and W Pugh. Finding bugs is easy. *SIGPLAN Not.*, **39**(12), 2004.
30. Thung F, Lucia, Lo D, Jiang L, Rahman F, Devanbu PT. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In ASE, 2012; 50–59.
31. SK Lukins, NA Kraft, and LH Etzkorn. Bug localization using latent Dirichlet allocation. *Information & Software Technology*, **52**(10):972–990, 2010.
32. Wang S, Lo D, Lawall J. Compositional vector space models for improved bug localization. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), 2014.
33. Moreno L, Treadway JJ, Marcus A, Shen W. On the use of stack traces to improve text retrieval-based bug localization. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), October 2014.
34. Hassan A. Predicting faults using the complexity of code changes. In Proceedings of the 31st International Conference on Software Engineering, ICSE ′09, 2009; 78–88.
35. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493.

36. Emam KE, Melo WL, Machado JC. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software* 2001; **56**(1):63–75.
37. Nagappan N, Ball T, Zeller A. Mining metrics to predict component failures. In Proceedings of the 28th International Conference on Software Engineering, ICSE ′06, 2006.
38. Marcus A, Poshyvanyk D, Ferenc R. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering* 2008; **34**(2):287–300.
39. T Zimmermann and N Nagappan. Predicting defects using network analysis on dependency graphs. In Proceedings of the 30th International Conference on Software Engineering, ICSE ′08, 531–540, 2008.
40. Poshyvanyk D, Gethers M, Marcus A. Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology* 2013; **21**(4):.
41. Poshyvanyk D, Guéhéneuc Y-G, Marcus A, Antoniol G, Rajlich V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering* 2007; **33**(6):.
42. Dit B, Revelle M, Poshyvanyk D. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering* 2013; **18**(2):277–309.
43. Gethers M, Dit B, Kagdi HH, Poshyvanyk D. Integrated impact analysis for managing software changes. *ICSE* 2012; 430–440.
44. Wang S, Lo D, Xing Z, Jiang L. Concern localization using information retrieval: An empirical study on linux kernel. In WCRE, 2011.
45. Dit B, Revelle M, Gethers M, Poshyvanyk D. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 2013; **25**(1):53–95.