Received January 15, 2022, accepted February 6, 2022, date of publication February 14, 2022, date of current version February 28, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3151395

FineCodeAnalyzer: Multi-Perspective Source Code Analysis Support for Software Developer Through Fine-Granular Level Interactive Code Visualization

ABDUL QAYUM[1], SAIF UR REHMAN KHAN[1], INAYAT-UR-REHMAN[1], AND ADNAN AKHUNZADA[2], (Senior Member, IEEE)
[1]Department of Computer Science, COMSATS University Islamabad (CUI), Islamabad 45550, Pakistan
[2]Faculty of Computing and Informatics, University Malaysia Sabah, Kota Kinabalu 88400, Malaysia

Corresponding authors: Adnan Akhunzada (adnan.akhunzada@ums.edu.my) and Abdul Qayum (eng.abdul.qayum@gmail.com)

ABSTRACT Source code analysis is one of the important activities during the software maintenance phase that focuses on performing the tasks including bug localization, feature location, bug/feature assignment, and so on. However, handling the aforementioned tasks on a manual basis (i.e. finding the location of buggy code from a large application) is an expensive, time-consuming, tedious, and challenging task. Thus, the developers seek automated support in performing the software maintenance tasks through automated tools and techniques. However, the majority of the reported techniques are limited to textual analysis where the real developers' concerns are not properly considered. Moreover, existing solutions seem less useful for the developers. This work proposes a tool (called as FineCodeAnalyzer) that supports an interactive source code analysis grounded on structural and historical relations at fine granular-level between the source code elements. To evaluate the performance of FineCodeAnalyzer, we consider 74 developers that assess three main facets: (i) usefulness, (ii) cognitive-load, and (iii) time efficiency. For usefulness concern, the results show that FineCodeAnalyzer outperforms the developers' self-adopted strategies in locating the code elements in terms of Precision, Recall, and F1-Measure of accurately locating the code elements. Specifically, FineCodeAnalyzer outperforms the developers' strategies up to 47%, 76%, and 61% in terms of Precision, Recall, and F1-measure, respectively. Additionally, FineCodeAnalyzer takes 5% less time than developers' strategies in terms of minutes of time. For cognitive-load, the developers found FineCodeAnalyzer to be 72% less complicated than manual strategies, in terms of the NASA Tool Load Index metric. Finally, the results indicate that FineCodeAnalyzer allows effectively locating the code elements than the developer's adopted strategies.

INDEX TERMS Source code analysis, bug localization, feature location, structural analysis, historical analysis, software maintenance, usefulness, bug assignment, interactive tool.

I. INTRODUCTION

Source code analysis is a core process performed by a software developer during the software maintenance phase [1]. For example, when developer intends to find a buggy code element or feature-related code elements from a software system, he needs to perform analysis on code. Mainly, software maintenance includes adding a new feature and fixing the existing one against a requested or reported by a user or other developer. A feature is requested a user observes that some functionality is missing and should be the part of software application. Whereas a bug/issue fixing request is mostly reported a user found a bug in the application. A feature is a functionality that can be observable from a software system [2]. Whereas a defect or flaw in a software system is known as a bug [3], which leads to producing an unexpected result or behavior in a software system [4].

Source Code analysis is easy when the software system is small, that is, having a few hundred to a few thousand

The associate editor coordinating the review of this manuscript and approving it for publication was Dongxiao Yu.

lines of code. A manual source code analysis is possible in this case. However, it becomes a worst, hectic and much time-consuming task when the software system is large, that is, many thousands to a millions line of codes [5]. In this case, practitioners demand some automatic tools or techniques for code analysis, that will support them to find the feature-related code elements or a buggy code element from this large software system. For example, Zhang *et al.* [6] suggested that developers spend more time fixing bugs than developing software. Whereas in large systems, like an eclipse, bugs/faults reported in a day may reach hundreds per day [7]. This shows that such activities take a long time to complete. For instance, Kim and Whitehead [8] reported in the PostgreSQL project that most bugs need 100 to 200 days to complete their resolution.

Consequently, the researchers proposed different techniques to perform automatic code analysis. Generally, software maintenance includes adding a new feature and debugging the existing code. In both cases, current research needs to perform an analysis of the source code. The general types of analysis are textual, static, dynamic, and historical [9].

A. TEXTUAL ANALYSIS

Textual analysis is the widely employed analysis in source code analysis, as reported by Dit *et al.* survey [9]. Textual analysis performs on the text present in the bug reports and the comments and identifiers present in the source code. In this way, textual analysis exploits the bug reports to formulate a query and retrieve the results by matching this query with available comments and identifiers in source code [10]. In such analysis, Information Retrieval (IR), Natural Language Processing (NLP), and Pattern Matching (PM) are the vital leveraging techniques [11], [12].

B. STATIC ANALYSIS

Static analysis is often based on the structure of code; hence, it is also known as structural analysis. In structural analysis, analysis has been performed based on call relations that exist between the source code elements. For example, there must be a relation between them if one method calls to another method or if one class is inheriting with another class. Moreover, in the static analysis, analysis has been performed on the code without executing the program [13].

C. DYNAMIC ANALYSIS

In dynamic analysis, the bug is identified or observed by the time of execution of the program. Generally speaking, dynamic analysis approach focused on bug localization by analyzing the data flow, control flow, execution traces, and breakpoints of the program. The dynamic methods examine the pass or fail execution traces of the program under a certain input condition and assign a suspicious score to each line of code [14].

D. HISTORICAL ANALYSIS

Finally, in historical analysis, analysis has been performed by mining the code change histories available in source code repositories. In historical methods, bug reports and code change histories were examined to find the bug location in a program [15]. Besides, some hybrid approaches have been observed in the literature based on the combination of above-mentioned textual analysis techniques [9].

Despite the presence of a large number of such solutions, Razzaq *et al.* [10] suggested that 172 different techniques for feature location have been presented in the literature; however, the software industry still minimaly employed the proposed feature localization techniques. For example, Parnin and Orso [17] and Kochhar *et al.* [18] conducted extensive surveys with the real developers to assess the relevance of the existing solutions in the industry. Their results suggested that the developers do not formally use any existing solution in the industry. This is because the presented techniques are evaluated with an effectiveness aspect. In other words, these techniques typically return a list of the source code elements which are ranked with their score of being buggy or related to a feature. These ranked lists are compared with the gold set, which is already known feature/bug-related code elements.

Hence, the effectiveness-based evaluations do not involve real developers into the loop at the time of evaluation. More recent research conducted with real developers, as proposed by Parnin and Orso [17] and Kochhar *et al.* [18], suggested that the techniques evaluated with effectiveness aspect do not meet the usefulness expectations of practitioners, and hence, are less adopted in software organizations. Also, to the best of our knowledge, there is no state-of-the-art source code analysis tool, general to feature and bug localization, openly available to be used for usefulness evaluation purposes.

Inspired by this, current research aims to support developer's performing source code analysis while doing some software maintenance-related task(s). This research presenting a UI based source code analysis tool that is more useful than developers' manual strategies to locate code. In addition, the presented tool also supports doing source code analysis in more than one context. Consequently, this research proposed a tool named FineCodeAnalyzer that is based on interactive code visualization. The proposed tool exploits the structural and historical relations that exists in the source code, and develops an interactive source code analysis interface that permits the interaction between developers and source code elements. This research has employed user studies and developer survey to evaluate the presented tool. This research has employed triangulated method [33] to evaluate FineCodeAnalyzer tool where two user studies, assessing the usefulness, are performed with developers. These studies are combined with two developer surveys, each performed after each user study. The purpose of these surveys is to assess the FineCode-Analyzer for cognitive-load purposes. The time efficiency of the presented tool is also assessed. The evaluation of the proposed tool has involved 74 developers/novices from industry/academia.

As the feature/bug location is similar to information retrieval where users locate/find the information they are interested in, to evaluate the proposed tool for usefulness aspect, this research has employed commonly used metrics in information retrievals, namely: precision, recall, and f-measure [29], [39] [30], [31]. To assess FineCodeAnalyzer for cognitive-load, the most frequently used metric TLX [34], [35]: NASA Task Load Index, is employed. Finally, to assess the proposed tool for time efficiency, this research has employed a number of minutes developers take to locate the code related to a bug or feature. The results of the presented evaluation indicate that developer performance improves using the presented tool, for usefulness, cognitive-load, and time-efficiency aspects of evaluations, as compared to developer own strategies without using the presented tool.

Overall, this research makes the following contributions:

- To allows source code analysis based on a hybrid approach using structural and historical analysis.
- To propose a tool that allows developers to search the methods information with various options including callers-callee methods relations, method name, developer name who committed this method, date on which the method was committed, and all the methods between a range.
- To develop a dataset based on historical and structural information of source code for a commonly used java software system.
- To provide a user interface-based support to trace the feature-related source code elements and find buggy code elements.

In summary, the proposed tool works at a fine source-code granularity level rather than a coarse level. In source code analysis, there are 4 levels of granularity including package level, class level, method level, and statement level. Package level and class level, after locating them, still require manual endeavor to locate related code elements because all the elements in a class are not necessarily part of a feature [16]. Whereas, at a fine-granular level, e.g., statement level, also not conveying the context of the statement until not go through the complete method that contains this particular statement. Thus, this research considers method-level granularity, as suggested by Zhang *et al.* [6] and Kochhar *et al.* [18].

The remaining paper is organized as follows: Section III describes the background and related work of the underlying study. Whereas Section IV describes the adopted research methodology. Section V illustrates the results and analysis of the proposed technique. In addition, Section VI describes about threats to validity and limitations of the study. Finally, Section VII concludes the current work and also outlines potential future research dimensions.

II. RESEARCH MOTIVATION

Source code analysis is a core process a software developer needs to perform in the software maintenance phase [1]. Code analysis is easy when a software system is small, i.e. having a few hundred to a few thousand lines of code. A manual source code analysis might be possible in this case. However, it becomes a worst, hectic and time-consuming task when software system is large, that is, many thousands to the millions line of codes, as suggested by Wong *et al.* [5]. In this case, practitioners seek some automatic tools or techniques for code analysis, that will support them to find the feature-related code elements or a buggy code element from such a large software system. In providing the automated support, the researchers proposed different techniques to perform automatic code analysis [15], [36] [38], [40].

Following are the three motivations for this research which are driven by the three different challenges:

Challenge 1: The existing techniques are less user-friendly in terms of requiring more cognitive effort from developers. Such techniques only present a ranked list of the code elements (mostly the names of code components, files, classes) which are hard to understand for the developers because this does not allow developers to interact with code using those elements as a pivot. For example, this might be possible that the bug/feature-related code lies near to the located elements and developers can reach those elements using their calls-relationships, co-change relationships in code history, other data, or control flow relationships.

Motivation 1: The first motivation is to effectively exploit such relationships in order to provide a more interactive source code analysis tool which will allow the developers to utilize the Callers-Callees relations that exists between methods, methods that change by the same developer in history, and so on.

Challenge 2: The current techniques mostly perform at a coarser-granular level. In other words, at component, file, or class level. Kochher *et al.* [18] suggested that conducted with the developers, that such a high-level of granularity is not a preferred choice of developers as these required developers to exert further effort in locating the buggy/feature-related code elements after locating the coarser component, file, or class.

Motivation 2: The second motivation is to address this concern of developers. In doing so, this research presents a tool that operates on method-level granularity which is least effort-intensive and more logical in terms of understanding the context of the located code.

Challenge 3: The existing techniques presented as a solution are mostly not assessed for the usefulness concerns of the practitioners and hence, their relevance to the software industry is questionable.

Motivation 3: Therefore, the third motivation of this research is to evaluate the presented tool involving real developers (or at-least novices) for the usefulness of the tool rather than only effectiveness which is currently the main employed evaluation aspect, presented in 84% of the studies in a systematic literature survey conducted by Razzaq *et al.* [10].

III. BACKGROUND AND RELATED WORK

This section discusses the background and related work of the presented research.

A. BACKGROUND

Source code analysis is a core process a software developer needs to perform in the software maintenance phase [1]. The developers perform source code analysis so frequently for many different reasons [32]. For example, bug localization, bug fixing, fault testing, feature location, feature/bug assignment, and so forth [53]. These are the common tasks for developers while software maintenance activities. A feature is a functionality that has visible characteristics and can be observable from a software system [2], [7]. Hence, the feature location is a process of locating all the locations that implement this observable functionality [8]. Besides, a defect or flaw in a software system is known as a bug [3], which leads to producing an unexpected result or behavior in a software system [4]. Hence, bug localization is the process of locating the buggy piece of code in a software system. Moreover, Feature/Bug assignment is the process of assigning a feature/bug to the relevant or appropriate developer [11].

Generally, the Feature or Bug Localization process is start when a user or developer request a feature or report a bug. A Feature or Bug report is a documented form of a report, which is created against a bug report or feature request, by a user and developers themselves [54]. The developers rely on these reports to fix the bugs. These reports are managed by Bug Tracking Systems [55]. These reports are also used in the formulation of queries. The queries are used to extract the feature or bug-related code elements from the software system [56]. The working of a query is it matches the keywords passed in a query with comments and identifiers present in a source code and return related found code elements.

Moreover, to manage code versions, Version Control Systems (VCS) have been used. VSC help to manage and merge code developed by different developers using different computers [37]. VCS also manages the historical information related to the source code. For example, it manages the information such as date of code commit, name of a developer who have committed the code, which component has been changed in this commit, what was the reason for change, and so forth. This information is also very helpful for developers to perform source code analysis.

1) SOURCE CODE ANALYSIS TYPES

The researchers proposed four general types of source code analysis that include textual, static, dynamic, and historical analysis [9]. All of the four types are used in feature/bug localization.

a: TEXTUAL ANALYSIS

The first component is textual analysis that is performed on the text present in the bug reports and the comments and identifiers present in the source code. In this way, textual analysis exploits the bug reports to formulate a query and retrieve the results by matching this query with available comments and identifiers in source code [10].

b: DYNAMIC ANALYSIS

In dynamic analysis, the bug is identified or observed by the time of execution of the program. Thus, this approach performs bug localization by analyzing the data flow, control flow, execution traces, and breakpoints of the program. The dynamic methods examine the pass or fail execution traces of the program under the certain input condition and assign a suspicious score to each line of code [14].

c: STRUCTURAL ANALYSIS

Structural Analysis (also known as Static Analysis) is often based on the structure of code. In structural analysis, analysis has been performed based on call relations that exist between the source code elements. For example, there must be a relation between them if one method calls to another method or if one class is inheriting with another class. For instance, maybe an output of a method could be an input for another method; therefore, the first method called the second method. Moreover, in the static analysis, analysis has when performed on the code without executing the program [13]. In other words, this technique extracted the structural relationships based on calling relationships that exists between source code without running the project.

d: HISTORICAL ANALYSIS

In historical analysis, analysis is performed by mining the code change histories available in source code repositories. In historical methods, bug reports, and code change histories are examined to identify the bug location in a program [15]. For instance, historical information that includes date of code commit, name of the developer who commit the code, which of the component has changed in this commit, what was the reason of change, and so forth. Such information is also very helpful for developers to perform source code analysis. For example, all the methods committed in a single commit may be part of a feature, or all the methods committed by the same author on the same date may be a feature [40], [41].

2) TECHNIQUES OF SOURCE CODE ANALYSIS

Generally, there are various techniques presented by researchers for source code analysis. For example, Information Retrieval (IR), Natural Language Processing (NLP), Knowledge Graph-based, Deep Learning, and Pattern Matching (PM) are the vital leveraging techniques [11], [12] [19]. Some are combinations of these such as Lam *et al.* [20] proposed a hybrid model of Deep Learning and Information Retrieval. Information Retrieval technique further proposed multiple models that include Vector Space Model (VSM), Latent Semantic Indexing (LSI), and Latent Dirichlet Analysis (LDA). These are the vital employing techniques for feature location.

3) LEVEL OF GRANULARITY IN A SOURCE CODE

When performing bug/feature localization, techniques return the ranked list according to the granularity level selected by the researcher. Source code analysis supports four types of granularities, which include module level, file level, class level, method level, and statement level [42]. Module-level granularity means ranked list contain modules of the project and each module consists of many files. On the other hand, file-level granularity means a file that may contains multiple classes. Whereas class level means source code element that consists of one class only. While in method-level granularity each element of the ranked list consists of a single method. Finally, statement-level granularity means a single line of code-based elements. Notice that the researchers prefer method-level granularity. This is because of the fact that method-level granularity is least effort intensive compared to the other source code analysis techniques [40].

B. RELATED WORK

Source code analysis is one of the most important activities in software maintenance [1]. Different studies proposed different techniques for source code analysis. Zhang *et al.* [57] proposed a deep learning-based bug localization model named as KGBugLocator, which is consisting of a knowledge graph and keywords supervised bi-directional attention mechanism. They employed knowledge graph nodes to represent code entities and directed edges to represent various relationships. Moreover, they have developed an NLP-based keywords attention mechanism to capture the code element corresponding to a bug report with more accurate semantics. They employed 4 open-source datasets including AspectJ, AWS, JDT, and Tomcat. To evaluate their model, they employed three metrics including Top-K, Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR). The results show that the proposed model performs better than state-of-the-art bug localization models. However, the reported study lacks in handling the semantic gap that exists between the different programming languages and natural languages present in bug reports. Moreover, they unable to deal with large and complex knowledge graphs. Most importantly, the existing model are non-interactive for the users.

Rodriguez-Prieto *et al.* [21] proposed a Java-based source code analysis platform (a compiler), namely ProgQuery. Their proposed platform is mainly depending on the Neo4j graph database. This platform allows developers to write a modified program for analysis in a declarative way. Besides, the compiler is also modified to compute 7 syntactic and semantic links of the code elements. This allows the developers to extract different types of knowledge/features by analyzing on the graph [22]. The authors evaluated with the programs collected by the CUP research group of the University of Edinburgh from GitHub [23]. Their research shows significant improvement in terms of time and scalability. However, their study is not providing a UI for interaction.

Abdelaziz *et al.* [24] presented the Graph4Code model based on knowledge graphs. Graph4Code is helpful in program search, refactoring, bug detection, and code understanding. The authors employed 1.3 million python files, collected from GitHub. Moreover, they considered various use cases to evaluate the performance of the proposed model. Finally, they mentioned that their proposed model attained promising results; however, their proposed model is non-interactive.

Rahman *et al.* [25] proposed a Statement level Bug Localization (SBL). Initially, the authors developed a Method Statement Dependency Graph (MSDG) based on related source code methods. Afterward, they developed a Node Predecessor-node Dependency Graph (NPDG) in which they combined the corpora of each node with their predecessor node present in MSDG. They employed Vector Space Model (VSM) to find the similarity between the bug report and node(s) present in NPDG and suggest buggy statements. They measured the effectiveness of their proposed system by using 3 open-source systems which are Eclipse, SWT, and Password Protector. Whereas, Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Top-N Rank metrics are used as evaluation metrics. They reported that their proposed model achieved sufficient accuracy of MRR and MAP metrics on given datasets. However, they only work on the statement level of granularity, which is not enough to convey the context of the bug to the developer. Hence, the developer still needs to go through the code to understand the context and fix the bug accurately.

Li *et al.* [26] proposed LaProb (a label propagation-based bug localization method) model, which is mainly based on two components including graph construction and label propagation. They utilized inter and intra-relations exists between bug reports and source files to build the graphs which they call BHG (Biparty Hybrid Graph). Further, to propagate a label, they developed an algorithm that works on BHG to extract the cascading relations that exist between bug reports and source files to localize the bugs. They performed a large-scale experiment on nine open-source systems which include SWT, AspectJ, Eclipse, ZXing, SEC, HIVE, HBASE, WFLY, and ROO. They employed Mean Average Precision (MAP), MRR, and Top-N rank metrics to measure the performance of the model. They reported significant improvements over state-of-the-art bug localization models. Although, their presented model localized bugs in a good way; however, they lacks in handling the scalability issue.

Huo *et al.* [27], proposed a novel approach named Control-flow Graph embedding based Convolutional Neural Network (CG-CNN) for bug localization. Their proposed approach is based on a control flow graph of source code, which is based on the structural and sequential dependency of source code. Further, they applied the multi-instance decomposition on the control flow graph to extract the semantic feature. The proposed model can learn the unified features from the control flow graph and perform better bug localization. The control flow graph may contain multiple paths, but the path having the most similarity with the bug report, labeled as a buggy file. They have employed 4 open-source datasets (which are: PDE, PU, JU, and Tomcat) in their research. They have measured the effectiveness of their model by using the three metrics which include, Top-K, MRR, and MAP. The experimental results indicated that the proposed model performs significantly better than the state-of-the-art models. This study aims to indicate the researchers exploit the control flow graphs to get improvement in bug localization techniques. The study is focused on only the control flow of the code, but some other flows also exist in code like data flow, which will fail this system. Hence, it is the main drawback of their proposed system. Moreover, CG-CNN is based on file-level granularity.

Rahman *et al.* [28] proposed Information Retrieval based framework named BugSRCH to create a model for bug localization and relevant project component search. The major contribution of this research was their proposed framework automatically selects the most suitable information retrieval model according to the assigned task. The authors employed four datasets that include Birt, Eclipse-UI, JDT, and SWT. They reported significant performance in terms of precision and recall. The focus of this research is only to select the appropriate technique for bug localization but not on bug's nature, characteristics and priorities. Which is the main limitation of this research.

Dilshener *et al.* [58] also presented an IR-based model which ranks the code element according to relevance with the bug report. Their proposed model works fast because it assigns relevancy score directly without looking past code. The employed datasets are AspectJ, Eclipse, SWT, ZXing, Tomcat, ArgoUML, Pillar 1 and Pillar 2. They evaluated their research by using MRR, MAP, and Top-N evaluation measures. They concluded significantly better performance than state-of-the-art models. Their tool generates the ranked list only which is not usable for many practitioners, as proposed by Parnin and Orso [17] and Kochhar *et al.* [18] research.

IV. RESEARCH METHODOLOGY

The objective of this research is to determine the impact of a UI-based more interactive, fine-level tool support in terms of its usefulness, cognitive load, and time efficiency while the developers performing software maintenance tasks. In other words, feature location and bug localization. The proposed tool is compared with developers' best using strategies to locate feature/bug related code elements. Specifically, the following three research questions are addressed in this research:

RQ-1: *How accurately does a developer locate buggy/ feature-related code elements using the presented tool in comparison to not using the presented tool?*

RQ-2: *Which approach (using the presented tool vs Developer's best using Strategy) leads to a lower cognitive effort, as perceived by developers while performing feature/bug location tasks?*

RQ-3: *Which approach (using the presented tool vs Developer's best using Strategy) is more time-efficient in terms of saving developers time while performing feature/bug location tasks?*

To address the devised research questions, we employed a triangulation method [33] to evaluate FineCodeAnalyzer. In this method, a developer survey is combined with a control experiment. The following sections discuss the design of each method and how data is gathered employing those methods.

A. PARTICIPANT CHARACTERISTICS

This research has recruited 74 participants from different companies. For sake of consistency, the same participants were involved in each method of data collection. The participants included 39 software developers and 35 Students/novices belongs to the 06 to 08 semesters. This research has only selected the participant who have coding experience in java. Hence, the proposed selected participants had at least one year of development experience using Java language. The following are the demographic information collected from the participants while evaluation.

Name — *Full Name of the Participant*
Current Role — *Current Designation of the Participant*
Age — *Current Age of the Participant*
Gender — *Information about Gender in terms of Male, Female and Prefer not to say*
Overall Experience — *Overall Programming Experience, in years*
Java Experience — *Java Development Experience, in years*
Code Maintenance Activities Frequency — *Information about how often Participants' involve in code maintenance*
Will you Recommend the proposed Solution? — *After using FineCodeAnalyzer (Proposed Tool), takes recommendations from participants about the tool*
Employee Email — *Email of the Participant*
Company Name — *Information about the company in which Participant is currently doing employment*
Address of the Company — *Information about Address of the company in which Participant is currently doing employment.*

Notice that this research also noted the time taken by each participant in locating the assigned feature/bug-related source code elements, mentioned in the performed two user studies in the following section.

B. EMPIRICAL DESIGN

This section illustrates the empirical settings adopted for assessing the performance of FineCodeAnalyzer.

1) USER STUDY 1

This step belongs to the control experiment performed with participants. Table 1 shows the features/bugs provided to the developers to locate from SWT system. These features/bugs are collected from SWT repository. In total, fifteen minutes of time is given for locating each feature/bug. This time is

TABLE 1. List of features/bugs provided to the developers.

Bug ID	Summary	Reported On	Reported By	Closed On	Closed By
351935	Modyfied GridLayout to work with index.	2011-07-13	Ludwig Moser	2011-07-13	Markus Keller
73228	We need API to set the timeout for tooltips showed via #setToolTipText API.	2004-09-03	Gunnar Wagenknecht	2004-09-09	Grant Gayed
402514	Resizing the display is valid when you change the resolution.	2013-03-06	Ivan Furnadjiev	2013-03-15	Silenio Quarti
530152	Would be great to have a Shell transparency feature in SWT.	2018-01-23	Ivan Morelli	2020-12-13	Eric Williams
288222	To detect location where we drop a file from application to local file system.	2009-09-01	Rencana Tarigan	2010-08-24	JuergandBaier
266114	Add export feature to sleak.	2009-02-25	Jacek Pospychala	2009-03-25	Jacek Pospychala
386267	Font: support of opentype localization feature	2012-08-01	Andre Saibel	2019-11-27	Lars Vogel
306931	Features tab in installation details window is (showing a white screen) taking time to load plugin.	2010-03-24	Nagaraju	2019-09-06	Paul Webster
97077	Table column should be multiline	2005-05-27 16	David Bourguignon	2017-07-28	David Bourguignon
121780	Browser popup does not keep user's web session	2005-12-21	KAR YEOW	2019-09-06	Eclipse Webmaster
244664	Text with swt.multi in scrolled composite eats the scroll wheel event	2008-08-20	Jerome Gout	2019-09-06	Alex Blewitt
237957	Styled text print options should include margins	2008-06-20	Peter Centgraf	2019-09-06	Eclipse Webmaster
205728	Dashed bored text stlye has various problems	2007-10-08	Dani Megert	2019-09-06	Markus Keller
217384	Ccombo should allow button image to customizeable	2008-01-31	Craig Salter	2019-09-06	Eclipse Webmaster
192956	[Ctabfolder] should investigate optimizing highlight drawing	2007-06-15	Kevin McGuire	2019-09-06	Eclipse Webmaster
170011	[Ctab folder] should feature request: allow writing text/image to tables folder	2007-01-09 15	Paul E. Keyser	2019-09-06	Eclipse Webmaster
157058	[Scrolledcomposite] setorigin method causes user click selection to not occur.	2006-09-12	Whitney Sorenson	2007-04-18	Carolyn MacLeod
125969	[Scrolled composite] swt layouts in scrolled composite in formtoolkit brokand	2006-01-31	Paul E. Keyser	2006-02-02	Steve Northover

decided based on the gained experience, while the multiple trials were performed before conducting this experiment. Then, opened the source code of SWT in Eclipse Integrated Development Environment (IDE) for the participants. The participants were asked to use their best-known strategy to locate the bugs/feature. For example, the developers may use the internet for searching some solution related to the feature/bug. In this case, they may search through the IDE to perform any sort of searching (i.e. file or text search). Additionally, they may use project explorer of call graphs built-in Eclipse IDE.

For each feature/bug, first descriptions were provided to the participants, and they were guided to fully understand the bug/feature report before proceeding to locate the bug/feature. Next, the developers were asked to make a list of methods related to the found bug/report. In case of all methods that belong to a class or file are buggy or related to the feature, they were asked to point the file or class. The experimentation also involves guiding the developers to consider the complete method as related to the feature/bug if only a portion of that they perceive as related. For example, if they find only a variable used in a method or a statement used in a method is related to the feature/bug and other part is not related, they were guided to consider the full method as related in such cases. Finally, the participants were asked to raise their hand once they complete locating each feature/bug. Otherwise, they will be interrupted once the pre-decided time to finish feature/bug localization is expired.

Once, the participants are ready for the experimentation. They start performing feature/bug location tasks for the given bug/feature report. The timer has been started to count the time a participant takes to finish the localization task(s). When the localization task is finished, the list of methods from the participants is collected as well as the finishing time is recorded. Consequently, the participants are directed to proceed to the next bug/feature.

To remove the biases of feature/bug characteristics (for example, the difference in the number of elements to locate, the difference in distance of the methods to locate from the main method, and so on.) that may impact their localization, we have used the randomization strategy [38]. Notice that the randomization strategy considers an equal number of features/bugs are evenly allocated to each of the participants. However, due to the non-availability of participants at a single time, this experiment is performed in an iterative manner with different groups of participants at different times.

2) SURVEY 1

After finishing User Study 1, we assessed the approach adopted by the participants regarding their effort excreted in locating feature/bug related elements. In other words, this survey is performed to assess the usefulness of the presented tool in terms of required effort as developers perceived when locating features/bugs using FineCodeAnalyzer compared to not using the presented tool. To perform this assessment, we have conducted a survey guided by NASA Task Load Index (TLX) measure [34], [35].

a: NASA TASK LOAD INDEX (TLX)

NASA-TLX is a multi-dimensional scale designed to obtain the workload estimates from one or more operators while performing a task or immediately afterward. The NASA Task Load Index (TLX) consists of six sub-scales: (i) Mental Demand, (ii) Physical Demand, (iii) Temporal Demand, (iv) Frustration, (v) Effort, and (vi) Performance. Notice that the TLX indexes represent the independent clusters of variables as highlighted in Table 2. The main assumption is that some combination of TLX dimensions is likely to represent the "workload" experienced by most people performing the localization tasks. These dimensions were selected after an extensive analysis of the primary factors that do (and do not) define the subjective experience of workload for different people performing a variety of activities ranging from simple laboratory tasks to flying an aircraft [34], [35]. Coincidentally, these dimensions also correspond to various theories that equate workload with the magnitude of the demands imposed on the operator, physical, mental, and emotional responses to those demands, or the operator's ability to meet those demands.

TABLE 2. TLX indexes and questions asked from the developers.

Number	Title	End Points	Description
1	Mental Demand	Low/High	How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, and so on)? Was the task easy or demanding, simple or complex, exacting or forgiving?
2	Physical Demand	Low/High	How much physical activity was required (e.g., pushing, pulling, turning, controlling, activating, and so on)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?
3	Temporal Demand	Low/High	How much time pressure did you feel due to the rate pace at which the tasks or task elements occurred? Was the pace slow or leisurely or rapid and frantic?
4	Frustration	Low/High	How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?
5	Effort	Low/High	How hard did you have to work (mentally or physically) to accomplish your level of performance?
6	Performance	Good/Poor	How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied you were with your performance in accomplishing the targeted goals.

Further, the data collected in this step is archived. Next, frequency analysis is performed on this data, as presented in Section V. In this evaluation, the selected participants are the same as described in Section IV (A). The following information has been collected from the participants.

Name — *Full name of Participant*
Mental Demand — *How much mental and perceptual activity was required in feature/bug localization with FineCodeAnalyzer in respect to Developer's best known/adopted Strategy?*
Frustration — *How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the feature/bug localization with FineCodeAnalyzer in respect to Developer's best known/adopted Strategy?*
Effort Required — *How hard did you have to work (mentally or physically) to accomplish your level of performance in feature/bug localization by using FineCodeAnalyzer in respect to Developer's best known/adopted Strategy?*

Physical Demand is not considered because since it is irrelevant to the targeted research context. In addition, Temporal and Performance is also not focused in this work because we have evaluate both of Temporal and Performance separately.

3) USER STUDY 2

This study is specifically performed to collect the data essential for the evaluation of the proposed tool. In terms of a study procedure, the task first gave a presentation to introduce the proposed tool and its features to the participants. This presentation included a twenty-minute live demo of how to use the proposed tools for one/two real features/bugs. Notice that the features/bugs used in the live demo were not considered during the experimentation process. To get accurate and comprehensive data, the participants were then asked to use the proposed tool to locate feature/bug-related code using the proposed tool. To remove the carry-over effect bias [66] in this step, it is important to make sure that the participants get different bugs/features to locate than in user study 1. However, we need to compare the performance using the proposed tool and not using the proposed tool. Thus, we did not change the dataset of user study 1. In other words, every participant got different feature/bug sets to locate but overall features/bugs were the same as of user study 1.

Similar to User Study 1, the participants were asked to create a list of methods they found related to the assigned feature/bugs. However, in addition to understanding the bug report and searching through the source code of SWT software system, the participants were also provided historic information about the bug and code elements. Specifically, this work has provided the date of the bug report actually reported on the issue tracking system. Moreover, for each code element, we have provided information about the dates when they change. Moreover, we have also provided information about the authors who changed the code elements.

The participants were directed to must use the proposed tool this time. Also, they were guided to follow the functionalities supported in tools to explore the code and history information when needed. Finally, they were informed that functionalities to filter based on date and author/developer information may reduce the required effort. Then, they started their localization tasks and we keep noting the timer. Similar to User study 1, they were informed to raise their hands when finished or will let them know when the given time is expired. After, each localization of feature/bug, we have collected the list of methods and noted the time taken by the participants in completing the given task.

4) SURVEY 2

As like survey 1, the same survey is performed with the proposed tool after participants finished user study 2. The same questions were considered (as described in Table 2) to assess the workload/cognitive effort participants excreted while performing localization tasks using the proposed tool. The data collected in this step is archived again for analysis purposes.

a: ANALYSES AND EVALUATION METRICS

To analyse the data collected and to compare the results of proposed tool with developers' own strategies, this research has employed different metrics for each research question.

b: RQ1: USEFULNESS EVALUATION

For RQ 1, we considered the commonly employed information retrieval metrics to compare the developers' performance when using the proposed tool and when not using it. The widely-employed metrics are described as follows:

1: Recall

Recall measures the fraction of feature/bug-related source code elements that are correctly identified by developers. Recall is defined in Equation 1:

$$\|CorrectElements \cap IdentifiedElements\| \\ -CorrectElements| \quad (1)$$

Equation 1 illustrates the formula to calculate the Recall measure. In this work, the Identified Elements are those elements, which are identified by a participant against a given task. Whereas, Correct Elements are those elements that are correctly identified.

2: Precision

Precision measures the fraction of developers' identified source code elements that are related to the feature/bug. It is formally defined using Equation 2, as follows:

$$\|CorrectElements \cap IdentifiedElements\| \\ -IdentifiedElements| \quad (2)$$

Equation 2 illustrates the formula to calculate the Precision measure. In current work, Identified Elements are those elements, identified by a participant against a given task. Whereas, Correct Elements are those elements, which are correctly identified.

Precision alone fails to measure the coverage of the results. In other words, finding all of the feature/bug-related elements, by ignoring not retrieved feature/bug-related elements. In contrast, Recall, by ignoring the incorrectly retrieved elements, fails to assess ranked-listing with lots of (distracting) false positives. F-Measure is another measure that gives a high value only in the case that both recall and precision values are high.

3: F-Measure

F-measure is a harmonic mean of recall and precision and is defined as in Equation 3:

$$2 \times \|Recall * Precision\|Recall + Precision| \quad (3)$$

Equation 3 illustrates the formula to calculate the F1-Measure. The identified Precision and Recall will help to identify the F1-Score.

Regarding the Usefulness concern, we employed all three measures, each having values in the range (0, 1). Specifically, we compared the recall, precision and f-measure values for the list of methods provided to us by the participants for both approaches, i.e. when using the proposed tool and when not using it.

c: RQ2: COGNITIVE-LOAD (DIFFICULTY) EVALUATION

To assess both approaches (i.e. when using the proposed tool vs. when not using it) for the effort required or cognitive load, (high/low) frequencies for NASA Task Load Index (TLX) related six attributes for each question are analyzed.

d: RQ3: TIME EFFICIENCY EVALUATION

The main hypothesis was that proposed tool would outperform the manual approach of the developer due to its single-analysis scheme and simplicity. To answer RQ3, we compared the time taken in minutes by the developers when localizing each feature/bug from SWT system, when using the proposed tool vs. when not employing the proposed tool.

Generally speaking, the biases can occur during time efficiency evaluation, which might be due to a change in the hardware of a system that includes a change in the computation, storage, capability of machines for the developers. To control this moderating factor, it is required to ensure that a participant who has performed user study 1 and user study 2, should work on the same machine. Moreover, we also make sure that while measuring the time efficiency, the developers must close all other irrelevant applications.

C. PROPOSED FRAMEWORK

This section illustrates the proposed framework that supports multi-perspective feature/bug localization.

1) DATASET AND BENCHMARK

We employ an SWT as a benchmark system. SWT system is an open-source software system based on 28702 commits, and 131 branches. SWT is an Eclipse-based Standard Widget Toolkit (SWT) framework that helps to develop Graphical User Interface (GUI) based applications in Java by using built-in UI based components of Operating System (OS) [60]. This research choose SWT system because this is one of the most commonly used system employed in bug/feature location [61], [62] [15], [63] [56], [64]. In addition, it has a built-in bug tracking system and easy to use from interaction perspective [65].

This work proposes a hybrid framework for source code analysis, as shown in Figure 1. The upper part of the framework is Code Driller (CD) that helps in extracting the dataset. CD has two further components including Code Parser (CP) and Code History Miner (CHM). Whereas the second part of framework is Interactive Code Maintainer (ICM) which helps to create interactive tool (FineCodeAnalyzer). Notice that we have extracted two types of code relations from the SWT system including Structural and Historical. We have developed a Code Parser to extract Structural Relationships and Code History Miner to extract Historical Relationships (Figure 1). Overall, we extracted seven tables considered as a proposed dataset. The details of the extracted datasets is shown in Table 3. The following section provides the details about the components of the proposed framework.

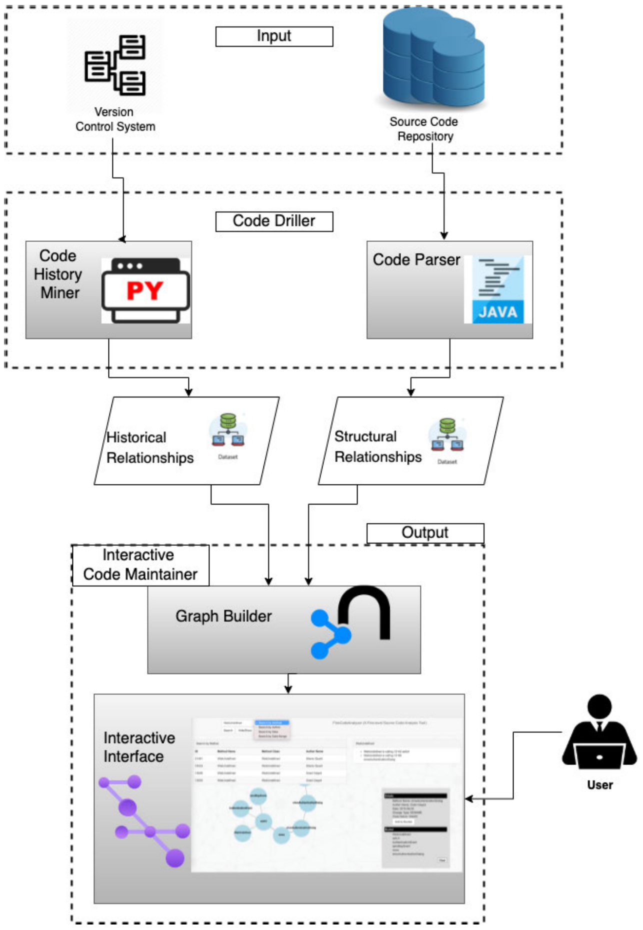

FIGURE 1. FineCodeAnalyzer framework.

TABLE 3. Dataset developed for this research.

Number	Name of Table	Number of Instances	Number of Attributes
1	Callers	15, 924	3
2	Callees	15, 924	3
3	Author Table	158	2
4	Method Table	31, 848	7
5	Changed Table	31, 848	5
6	Commit Table	31, 848	4
7	Modified Methods	50, 010	7

2) CODE PARSER

To extract Structural Relationships between code elements, we have developed a Java based Code Parser (CP). Code Parser takes a source code repository as an input and produces all the methods in a system. Initially, Code Parser extracts 80,726 methods. We used these methods to create Callers (i.e. a method who is the caller of some other method) and Callees (i.e. a method who is being called by some other method). We further drops the system built-in methods and create Callers and Callees methods from the considered methods and obtained 28,760 Callers and Callees.

Afterward from the Callers and Callees, we further drops all such methods who are not caller or callee of any other method(s). Consequently, we attained 15,924 Callers and

Callees. The final datasets are based on three attributes: (i) Class Name, (ii) Method Name, (iii) Method Parameters. Further, we considered three attributes to get the unique methods. This is due to the fact that if only method names were taken, then this would exclude all of the overloaded and overridden methods. Finally, the proposed framework combines the Callers and Callees tables based on their call relation and makes a final matched table. This table only contains method id and method name. Consequently, we attained Structural Relationships based methods from the Code Parser.

3) CODE HISTORY MINER

To extract the historical relations in the source code elements, we developed a Python-based Code History Miner (CHM). It takes a version control system-based library as input and produced historical-based relations between methods as an output (Figure 1). By using Code History Miner, the proposed framework produces five different tables that contain various types of historical information related to methods as extracted from Code Parser. The following are the descriptions of tables:

- **Author Table:** This table contains data related to authors, such as, Author Name and Author ID. In total, 158 Authors are of the proposed subject system.
- **Method Table:** This table contains information related to the method. The information includes Class from this belongs to, Method Name, Method Parameters, Change Type, Date of develop this method, and Author ID from Author table.
- **Changed Table:** This table contains information about the changes that have been done in the methods. The attributes of this table are Commit ID, Author ID, Method ID, Date of change, and Change Type.
- **Commit Table:** This table contains information related to commits, such as, Commit Hash, Author name who did a particular commit, and date of a particular commit.
- **Modified Methods:** This table contains information related to modified methods only.

We have used both Structural and Historical based datasets, described above. Both of these types datasets are loaded in Neo4j to create proposed databases, Table 3 illustrates the dataset used by this research. Neo4j is a graph database having advantages over regular databases. Such as it provides flexible, easily scalable, responsiveness, and many other benefits. This research has used Neo4j to create the developed such databases for this research. Moreover, this research has used Neo4j to create a relation between all of the proposed tables.

4) GRAPH BUILDER

This component of the proposed model takes structural and historical relationships as an input and then created a graph (Figure 1). This graph is based on two components: (i) vertices and (ii) edges. Vertices are presenting the source code elements (methods in our case) of the project used as an input and edges are presenting the correlations between the considered code components. This research employed vertices and edges in creating a graph oriented network from datasets (code elements). Thus, graph networks provides an easy analysis on the data, even on a complex networks [68] while expressing the software applications in a connected graphs shapes [69]. That includes nodes and edges which helps to create different types of networks, for example code classes networks [70].

Graph Builder is based on the Neo4j platform, which has a D3 (a JavaScript) library on its top side. A Cypher Query Language is used to perform operations on the dataset created in Neo4j. It is essential to create a connection between D3 and Cypher to create graphs. This connection is made by using Python language. Overall, this component takes relationship(s) data as an input and creates a graph grounded on source code components, and also highlights the relationship between them.

5) INTERACTIVE INTERFACE

This component is based on a User Interface (UI) that will allow developers to interact with it and perform many types of tracking on source code components (Figure 1). For example, a developer will be able to see all the code components which were changed in a specific single commit. This will help the developer to find a particular feature. Likewise, the developer will be able to see which particular developer had changed the particular code components. This will help developers to assign the bug to the same developer who had fixed this bug before. In this way, this interface will help developers to localize bugs by tracking the code entities. The proposed tool can be download from the link.[1]

V. RESULTS AND ANALYSIS

This section presents the results and analysis with respect to each of the devised research questions.

RQ1: How accurately does a developer locate buggy/feature related code using the developed tool in comparison to Developer's best known/adopted strategy?

To answer RQ1, we performed a usefulness evaluation of FineCodeAnalyzer and Developer's best known/adopted strategy. The following section provides the results and analysis about usefulness evaluation.

A. USEFULNESS

This is the first evaluation aspect performed to evaluate the usefulness viewpoint of FineCodeAnalyzer. In this evaluation aspect, we conducted 74 developers based assessments. In this evaluation, each of the developers was assigned a feature or bug and was asked to make a list of all the methods related to a given feature or bug. In addition, each developer was assigned an equal number of times that is, 15 minutes,

[1]https://drive.google.com/drive/folders/15UiW5xuW6KB0-EK-XlXCPq2a4nyncXmc?usp=sharing

to achieve the given task. This research has have collected the following information against each given task:

- Developer's name
- Assigned feature/bug id
- List of methods that are extracted by developer
- Time on which the developer has completed the task

Based on the extracted methods, we have calculated some further information such as Relevant list methods and unrelated list methods based on gold-set elements. Already known feature/bug related code elements are known as Gold-sets [52]. Finally, based on the attained values in this work, we assess the performance of FineCodeAnalyzer in terms of Precision, Recall, and F1-Measure.

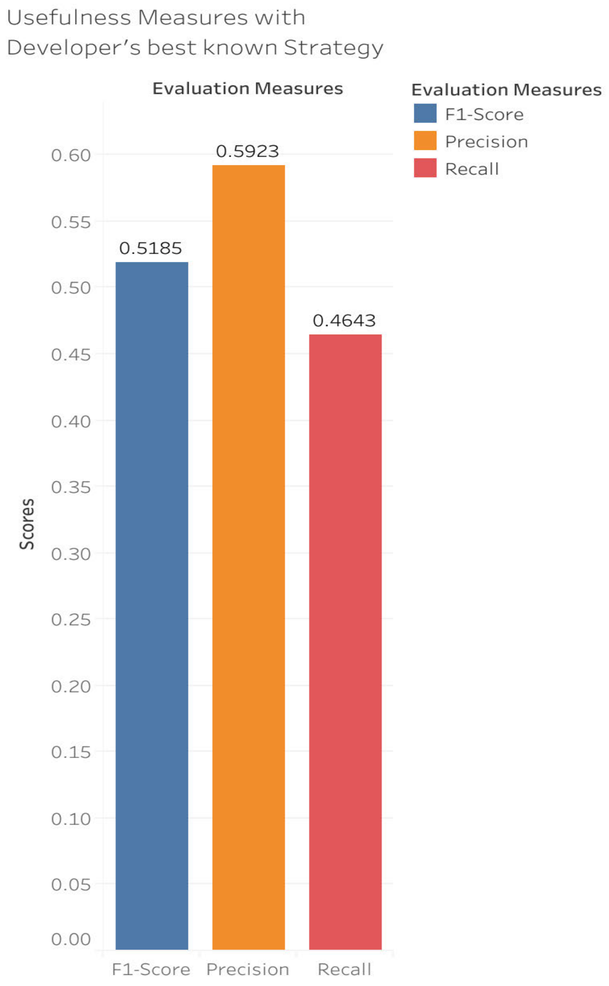

FIGURE 2. **Precision, Recall and F1-Measure results of developer's best known/adopted strategy.**

Figure 2 illustrates the obtained results regarding the usefulness evaluation with the developer's best-known feature/bug localization strategy. It can be clearly observed from Figure 2 that the Developer's best strategy has achieved 59%, 46% and 52%, Precision, Recall, and F1-Measure, respectively. However, using FineCodeAnalyzer, the developer's attained 87%, 81% and 84%, Precision, Re-call, and

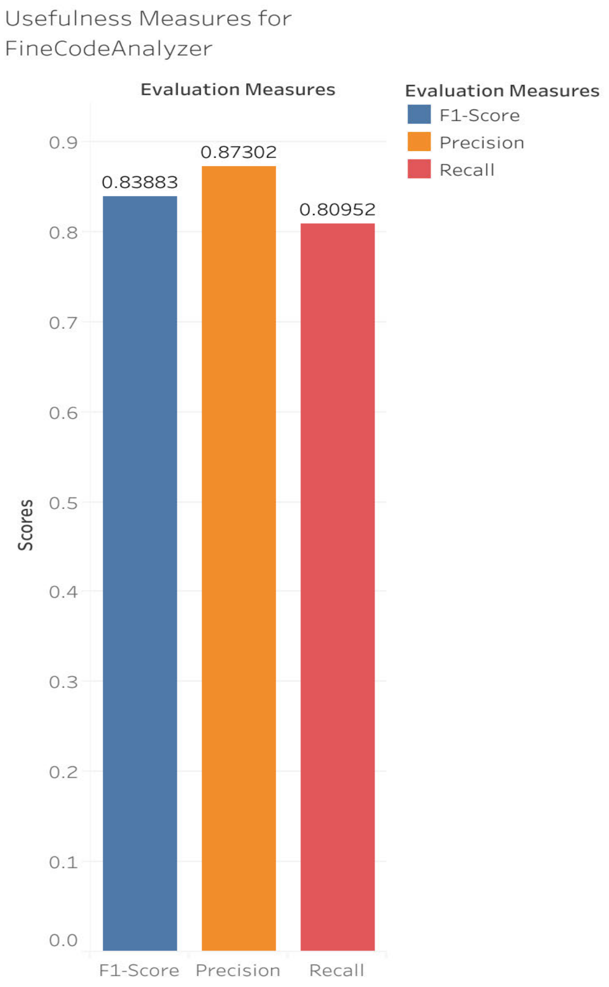

FIGURE 3. **Precision, Recall and F1-Measure results of FineCodeAnalyzer.**

F1-Measure, respectively, as shown in Figure 3. Evidently, FineCodeAnalyzer outperformed the developer's own methods of feature/bug location for 28%, 35% and 32% in Precision, Recall, and F1-Measure, respectively (as indicated in Figure 4).

RQ2: Which approach (using the developed tool vs developer's best known/adopted strategy) leads to a lower cognitive effort, as perceived by developers while performing feature/bug location tasks?

To answer RQ2, we performed Cognitive-Load based evaluation of FineCodeAnalyzer and Developer's best known/adopted strategy. The results and analysis on Cognitive-Load evaluation are described as follows:

B. COGNITIVE-LOAD

This is the second evaluation measure employed to assess the performance of the proposed tool. We measured the cognitive load of participants by employing the NASA based Task Load Index (NASA-TLX). The NASA-TLX is a multi-dimensional scale designed to obtain workload estimates from one or more

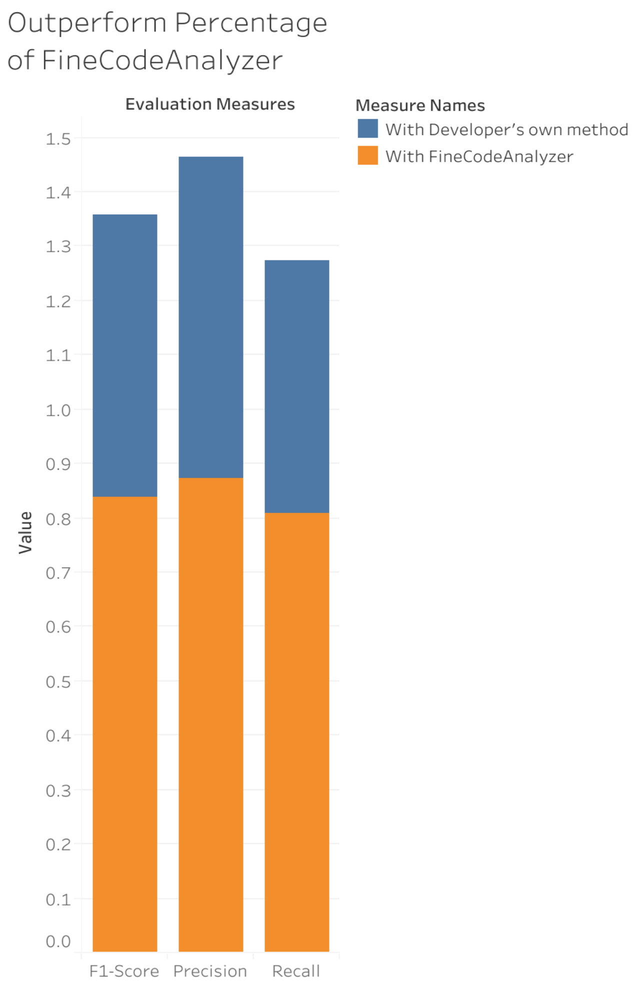

FIGURE 4. **Percentage of outperformed results of FineCodeAnalyzer than developer's best known/adopted strategy.**

operators while they are performing a task or immediately afterward. The NASA Task Load Index (TLX) consists of six sub-classes that represent somewhat independent clusters of variables including Mental, Physical, and Temporal Demands, Frustration, Effort, and Performance. Notice that we assumed that some combination of the considered dimensions is likely to represent the ''workload'' experienced by most people performing most tasks. The dimensions were selected after an extensive analysis of the primary factors that do (and do not) define the subjective experience of workload for different people performing a variety of activities ranging from simple laboratory tasks to flying an aircraft [34], [35].

Coincidentally, these dimensions also correspond to various theories that equate workload with the magnitude of the demands imposed on the operator, physical, mental, and emotional responses to those demands, or the operator's ability to meet those demands. We have calculated the Cognitive Load (from collected data) in three perspectives including Mental, Frustration, and Effort Demand. For this purpose, we have created a Google form, and asks each of the above-mentioned question from the participants after finishing both of the planned surveys. However, we did not calculate the Physical demand since it is irrelevant to the current work. In addition, the performance and Temporal Demand were not calculated because we have calculated these concerns separately. The questions and results of Cognitive-Load evaluations have been presented in Figures 5, 6 and 7.

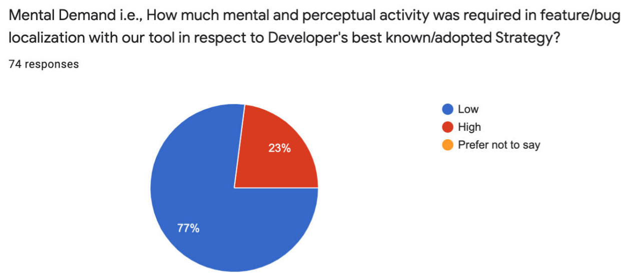

FIGURE 5. **Mental Demand of FineCodeAnalyzer in comparison to not using FineCodeAnalyzer.**

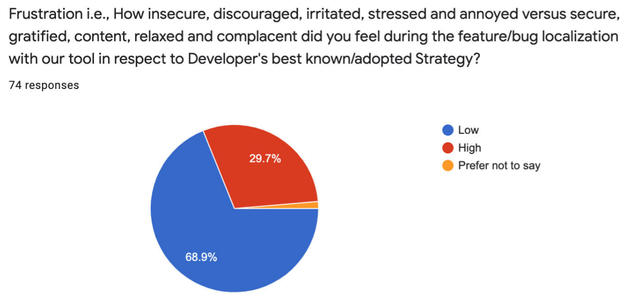

FIGURE 6. **Frustration level on FineCodeAnalyzer in comparison to not using FineCodeAnalyzer.**

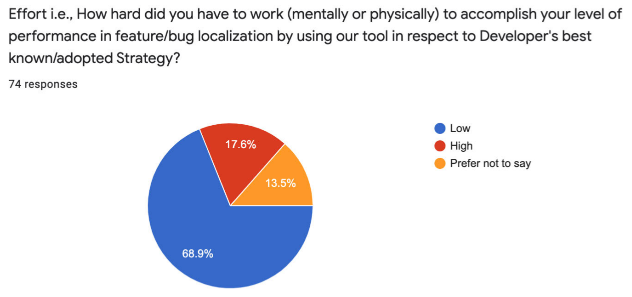

FIGURE 7. **Effort required in FineCodeAnalyzer in comparison to not using FineCodeAnalyzer.**

Figure 5 illustrates the Mental Demand of FineCodeAnalyzer in respect to the participant's preferred method of feature/bug localization. The result indiactes 77% participants feel FineCodeAnalyzer requires low Mental demand. While 23% participants feel that the prospoed framework demands a high level of Mental power. On the other hand, 68.9%

and 29.7% participants feel low and high frustration, respectively, in finding feature/bug by using FineCodeAnalyzer (Figure 6). However, 1.4% did not comment on frustration. Finally, in Figure 7, 68.9% participants feel low effort level is required in finding feature/bug by using FineCodeAnalyzer. Whereas 17.6% feel a high effort level is required in finding feature/bug by using FineCodeAnalyzer. However, 13.5% participants did not suggest any effort level required by FineCodeAnalyzer.

RQ3: Which approach (using the developed tool vs developer's best known/adopted strategy) is more time-efficient in terms of saving developers time while performing feature/bug location tasks?

To answer RQ3, we performed time evaluation of FineCodeAnalyzer and the Developer's best known/ adopted strategy. The results and analysis on time evaluation are presented in the following section.

C. TIME EFFICIENCY

This is the second evaluation measure performed to assess the time efficiency of the proposed tool. In this evaluation measure, we assigned a feature or bug to each of the participant and ask them to make a list of the assigned feature/bug-related source code elements (i.e. methods). To complete the task, we assigned a constant number of minutes to each participant (i.e 15 minutes). The given time (i.e. each developer taken to complete the task) has been noted. The information of each developer with the feature/bug id they have solved with respect to time has been shown in Figure 8.

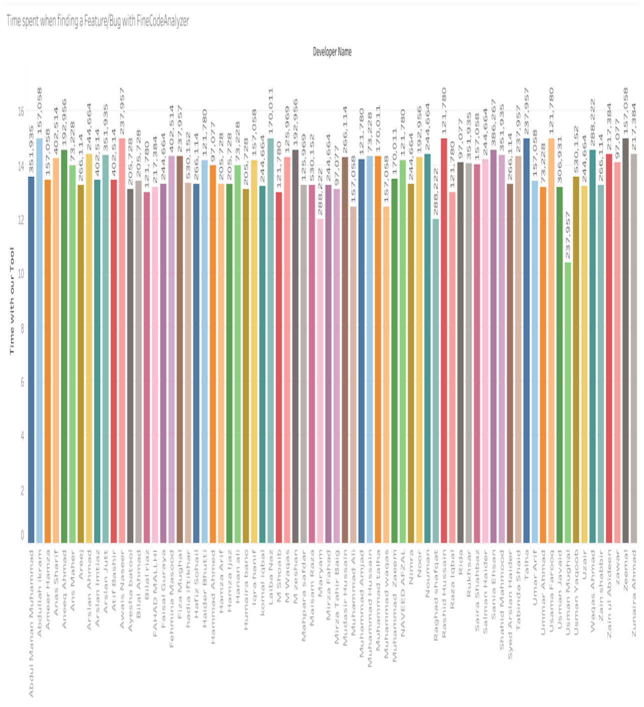

FIGURE 8. Time efficiency of FineCodeAnalyzer with each participant.

The results of the time evaluation of FineCodeAnalyzer are also compared with the Developer's best known/adopted strategy. Figure 9 illustrates the time efficiency of finding feature/bug-related methods with FineCodeAnalyzer vs Developer's best-known Strategy. The left part of figure shows the time a developer took to find feature/bug related code methods with Developer's best known Strategy presented tool (Figure 9). Whereas the right side of the graph shows the time a developer took to find feature/bug related code methods with the presented tool. The darker color indicates the highest time, whereas, the lighter color indicates the lower time, as shown in the Figure legends. From Figure 9, it can be clearly that finding a feature/bug using the presented tool required significantly less computational time against not using the proposed tool. Consequently, FineCodeAnalyzer effectively performed all of 74 tasks in 1,018 minutes. However, the same tasks performed by the developers taken 1,058 minutes sing their most convenient technique.

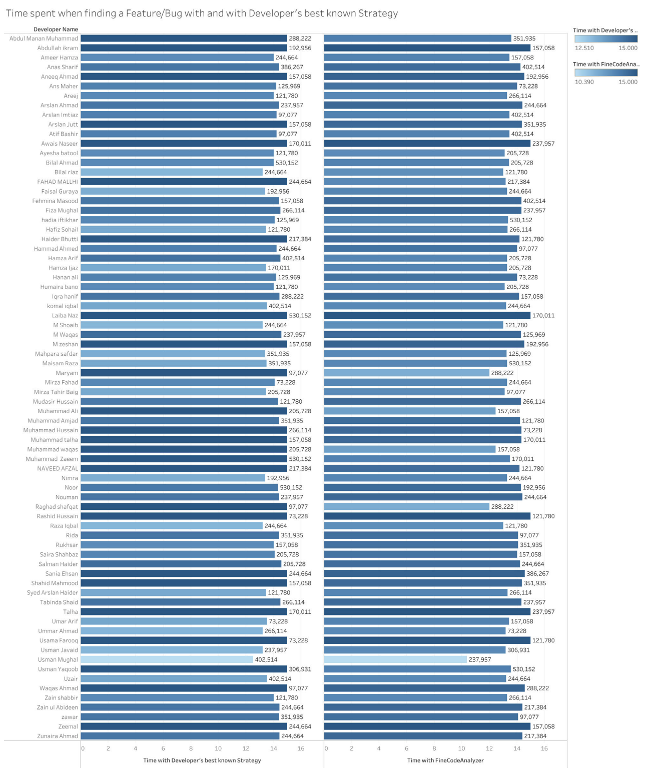

FIGURE 9. Time efficiency with FineCodeAnalyzer vs Developer's best known/adopted strategy.

VI. THREATS TO VALIDITY AND LIMITATIONS

As with any empirical study, several threats to validity posed to the current work. The potential threats are discussed in terms of internal, construct, and external validity.

A. INTERNAL VALIDITY

The developers may have learnt some experience on features/bugs while locating using some preferred strategy before locating them using the proposed tool. This experience gaining mechanism is called as carry over effect [66], and may bias towards the later treatment. To address this concern, we have used a randomization strategy [67], where each participant is provided a different feature/bug in each of both trials (i.e. without vs. with proposed tool). However, the experience gains of watching the overall source code of SWT (not for specific feature or bug) has not been controlled in this research and may have impacted the results.

To assess the tool for industrial concern, we employed the developers. However, due to the non-availability of highly experienced developers, this research has employed participants with an average experience of 3 years. The involvement of more experienced developers may impact the results of this research.

The relative performance (i.e. percentage of the difference) between the using the presented tool and not using presented tool is calculated after rounding the values to two decimal points for scores of evaluation metrics for all of the considered evaluation aspects. The relative performance calculated using different numbers of decimals might change the results slightly. The researchers interested in checking the results with a different number of decimals are invited to use the intermediate results provided with this research.

B. CONSTRUCT VALIDITY

The main construct validity issue in the software maintenance-related tools and techniques evaluation is the benchmark creation used for evaluation. In this case, the benchmarks are created for SWT, a frequently used system, where the gold-set is already available from the existing studies. However, the frequent adoption of the system/benchmark does not guarantee the accuracy and completeness of the benchmark. Hence, like the other studies in this field, the result of this research may also change with the change of benchmark employed.

This research has employed precision, recall, and f-measure metrics, which are the commonly used metrics in information retrievals. These all metrics capture a single concern of evaluation: That is, accuracy in locating feature/bug related code elements. However, these metrics may not fully capture the developer's intentions to locate the code elements. For example, developer may be interesting in locating a variable name in a located method rather than the complete method. In this work, we have to focused on one granularity, and we selected method level in the targeted research context. This decision is driven by Kochhar's work [18] performed with the developers.

C. EXTERNAL VALIDITY

Because FineCodeAnalyzer is evaluated for usefulness aspect rather than effectiveness. Due to this reason, we do not compared FineCodeAnalyzer tool with the existing baseline state-of-the-art techniques VSM, LSI or LDA [43], [44], [45], which are evaluated using effectiveness aspect and has not adopted in the industry, as suggested by the reported studies [46], [47], [48], [49]. This may have impacted on the generalizability of the presented tool for effectiveness aspect. However, it can be argued that our goal was to focus on the developer's concerns rather than the researchers.

Also, this research has only used SWT system which is an open-source Java-based system. Although, we have employed one of the commonly used systems in the field [46], [50], [14], [51]. This may limit the generalizability of the findings to only Java-based non-commercial software systems.

Finally, the participants involved in this research are mostly single city-based. Involving the participants from multiple cities, and even from multiple countries, and of different demography may produce a different result. Also, consider that this research is limited to only bug/feature location activities under software maintenance and one cannot generalize to other software maintenance activities such as bug severity detection.

VII. CONCLUSION AND FUTURE WORK

This work proposed an interactive code visualization-based tool names as FineCodeAnalyzer. The proposed tool is based on hybrid analysis of software system source code that exploits the structural and historical relations that exist in the source code. Moreover, the proposed tool works on a fine granularity level. In other words, FineCodeAnalyzer adopted a method level of granularity to save the developers' additional efforts to find the related code elements from large files. In addition, FineCodeAnalyzer is a generic tool that can be used in different contexts (tasks) related to software maintenance. We also evaluated the performance of FineCodeAnalyzer by involving 74 developers including senior students (i.e. considered as novices in this research context) and industrial practitioners. For example, FineCodeAnalyzer is assessed for feature and bug localization tasks. The results indicated that FineCodeAnalyzer significantly outperformed the manual strategies of the developers, which are usually employed while performing bug/feature localization-related tasks.

Following are the key findings of the proposed tool, which are outlined based on the attained results:

1) The proposed tool, FineCodeAnalyzer, outperformed the developers' manual strategies in locating buggy and feature-related elements in terms of usefulness, which is measured using precision, recall, and f-measure.
2) FineCodeAnalyzer performed better than the developers' manual strategies in locating code elements in terms of required cognitive/mental effort.
3) FineCodeAnalyzer attained better results about the execution time than the developers' manual strategies when locating code elements for features or bugs.

The practitioners can adopt the presented FineCodeAnalyzer in performing several software maintenance tasks where source code analysis is required. In contrast, the practitioners can employ FineCodeAnalyzer for other software maintenance related activities.

In the future, Deep Learning based algorithm can be adapted to further improve the performance of FineCodeAnalyzer. Surely, the adoption of a deep learning-based algorithm can be helpful in making a fully automatic bug/feature localization tools. Moreover, another potential research dimension could be to incorporate the dynamic analysis in the proposed tool. Consequently, it would support in fully automating the process for the developers in terms of accurately locating the source-code elements.

ACKNOWLEDGMENT

The authors are grateful to Software Reliability Engineering Group (SREG) members at COMSATS University Islamabad (CUI) for their continuous support and feedback throughout this research work. Moreover, they appreciate the experts who have participated in the conducted survey and provided their valuable responses.

REFERENCES

[1] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction," in *Advances in Intelligent Systems and Computing*. Cham, Switzerland: Springer, 2020, pp. 165–175.

[2] A. Razzaq, A. Le Gear, C. Exton, and J. Buckley, "An empirical assessment of baseline feature location techniques," *Empirical Softw. Eng.*, vol. 25, no. 1, pp. 266–321, Jan. 2020.

[3] J. Uddin, R. Ghazali, M. M. Deris, R. Naseem, and H. Shah, "A survey on bug prioritization," *Artif. Intell. Rev.*, vol. 47, no. 2, pp. 145–180, Feb. 2017.

[4] N. K. Nagwani and S. Verma, "Software bug classification using suffix tree clustering (STC) algorithm," *Int. J. Comput. Sci. Technol.*, vol. 2, no. 1, pp. 36–41, 2011.

[5] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[6] W. Zhang, Z. Li, Q. Wang, and J. Li, "FineLocator: A novel approach to method-level fine-grained bug localization by query expansion," *Inf. Softw. Technol.*, vol. 110, pp. 121–135, Jun. 2019.

[7] W. Zhang, S. Wang, and Q. Wang, "BAHA: A novel approach to automatic bug report assignment with topic modeling and heterogeneous network analysis," *Chin. J. Electron.*, vol. 25, no. 6, pp. 1011–1018, Nov. 2016.

[8] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. Eur. Softw. Eng. Conf. Found. Softw. Eng. Symp.*, Amsterdam, The Netherlands, 2009, pp. 111–120.

[9] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *J. Softw., Evol. Process*, vol. 25, no. 1, pp. 53–95, 2013.

[10] A. Razzaq, A. Wasala, C. Exton, and J. Buckley, "The state of empirical evaluation in static feature location," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 1–58, Feb. 2019.

[11] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The impact of classifier configuration and classifier combination on bug localization," *IEEE Trans. Softw. Eng.*, vol. 39, no. 10, pp. 1427–1443, Oct. 2013.

[12] A. Mahmoud and G. Bradshaw, "Estimating semantic relatedness in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 1–35, Dec. 2015.

[13] S. Wang and D. Lo, "AmaLgam+: Composing rich information sources for accurate bug localization," *J. Softw., Evol. Process*, vol. 28, no. 10, pp. 921–942, 2016.

[14] Z. Shi, J. Keung, K. E. Bennin, and X. Zhang, "Comparing learning to rank techniques in hybrid bug localization," *Appl. Soft Comput.*, vol. 62, pp. 636–648, Jan. 2018.

[15] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Inf. Softw. Technol.*, vol. 82, pp. 177–192, Feb. 2017.

[16] C. Tantithamthavorn, A. Ihara, H. Hata, and K. Matsumoto, "Impact analysis of granularity levels on feature location technique," in *Requirements Engineering*. Berlin, Germany: Springer, 2014, pp. 135–149.

[17] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proc. Int. Symp. Softw. Test. Anal.*, Toronto, ON, Canada, 2011, pp. 199–209.

[18] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Saarbrücken, Germany, Jul. 2016, pp. 165–176.

[19] S. Polisetty, A. Miranskyy, and A. Başar, "On usefulness of the deep-learning-based bug localization models to practitioners," in *Proc. 15th Int. Conf. Predictive Models Data Anal. Softw. Eng.*, Recife, Brazil, Sep. 2019, pp. 16–25.

[20] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension (ICPC)*, Buenos Aires, Argentina, May 2017, pp. 218–229.

[21] O. Rodriguez-Prieto, A. Mycroft, and F. Ortin, "An efficient and scalable platform for Java source code analysis using overlaid graph representations," *IEEE Access*, vol. 8, pp. 72239–72260, 2020.

[22] F. Ortin, O. Rodriguez-Prieto, N. Pascual, and M. Garcia, "Heterogeneous tree structure classification to label Java programmers according to their expertise level," *Future Gener. Comput. Syst.*, vol. 105, pp. 380–394, Apr. 2020.

[23] (Dec. 6, 2021). *Toegang Verkry*. [Online]. Available: http://groups.inf.ed.ac.U.K./cup/javaGithub

[24] I. Abdelaziz, J. Dolby, J. P. McCusker, and K. Srinivas, "Graph4code: A machine interpretable knowledge graph for code," 2020, arXiv:2002.09440.

[25] S. Rahman, M. M. Rahman, and K. Sakib, "A statement level bug localization technique using statement dependency graph," in *Proc. 12th Int. Conf. Eval. Novel Approaches Softw. Eng.*, Porto, Portugal, 2017, pp. 171–178.

[26] Z. Li, Z. Jiang, X. Chen, K. Cao, and Q. Gu, "Laprob: A label propagation-based software bug localization method," *Inf. Softw. Technol.*, vol. 130, Feb. 2021, Art. no. 106410.

[27] X. Huo, M. Li, and Z.-H. Zhou, "Control flow graph embedding based on multi-instance decomposition for bug localization," in *Proc. Conf. AAAI Artif. Intell.*, Apr. 2020, vol. 34, no. 4, pp. 4223–4230.

[28] M. M. Rahman, S. Chakraborty, G. Kaiser, and B. Ray, "Toward optimal selection of information retrieval models for software engineering tasks," in *Proc. 19th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Cleveland, OH, USA, Sep. 2019, pp. 127–138.

[29] B. Sisman, S. A. Akbar, and A. C. Kak, "Exploiting spatial code proximity and order for improved source code retrieval for bug localization," *J. Softw., Evol. Process*, vol. 29, no. 1, Jan. 2017, Art. no. e1805.

[30] R. Müller and U. Eisenecker, "A graph-based feature location approach using set theory," in *Proc. 23rd Int. Syst. Softw. Product Line Conf.*, Paris, France, Sep. 2019, pp. 88–92.

[31] G. K. Michelon, L. Linsbauer, W. K. G. Assunção, S. Fischer, and A. Egyed, "A hybrid feature location technique for re-engineeringsingle systems into software product lines," in *Proc. 15th Int. Work. Conf. Variability Model. Softw.-Intensive Syst.*, 2021, pp. 1–9.

[32] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, and F. Sarro, "A survey on machine learning techniques for source code analysis," 2021, *arXiv:2110.09610*.

[33] D. S. Triangulation, "The use of triangulation in qualitative research," *Oncol. Nursing Forum*, vol. 41, no. 5, pp. 545–547, 2014.

[34] S. G. Hart and L. E. Staveland, "Development of NASA-TLX (task load index): Results of empirical and theoretical research," in *Advances in Psychology*. Amsterdam, The Netherlands: Elsevier, 1988, pp. 139–183.

[35] S. G. Hart, "NASA-task load index (NASA-TLX); 20 years later," in *Proc. Hum. Factors Ergonom. Soc. Annu. Meeting*, Oct. 2006, vol. 50, no. 9, pp. 904–908.

[36] R. Gharibi, A. H. Rasekh, M. H. Sadreddini, and S. M. Fakhrahmad, "Leveraging textual properties of bug reports to localize relevant source files," *Inf. Process. Manage.*, vol. 54, no. 6, pp. 1058–1076, Nov. 2018.

[37] B. Ledel and S. Herbold, "Broccoli: Bug localization with the help of text search engines," 2021, *arXiv:2109.11902*.

[38] F. Pérez, J. Font, L. Arcega, and C. Cetina, "Collaborative feature location in models through automatic query expansion," *Automated Softw. Eng.*, vol. 26, no. 1, pp. 161–202, Mar. 2019.

[39] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4BL: Reproducibility study on the performance of IR-based bug localization," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Amsterdam, Netherlands, Jul. 2018, pp. 61–72.

[40] M. Chochlov, M. English, and J. Buckley, "A historical, textual analysis approach to feature location," *Inf. Softw. Technol.*, vol. 88, pp. 110–126, Aug. 2017.

[41] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proc. 22nd Int. Conf. Program Comprehension*, Hyderabad, India, 2014, pp. 53–63.

[42] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "SLDeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert Syst. Appl.*, vol. 147, Jun. 2020, Art. no. 113156.

[43] O. P. Sangwan, "Review of text mining techniques for software bug localization," in *Proc. 9th Int. Conf. Cloud Comput., Data Sci. Eng.*, Jan. 2019, pp. 208–211.

[44] K. Sharma and T. Sharma, "Software bug localization using pachinko allocation model," in *Proc. 3rd Int. Conf. Comput. Sustain. Global Develop.*, 2016, pp. 3603–3608.

[45] Y. Wang, Y. Yao, H. Tong, X. Huo, M. Li, F. Xu, and J. Lu, "Bug localization via supervised topic modeling," in *Proc. IEEE Int. Conf. Data Mining (ICDM)*, Singapore, Nov. 2018, pp. 607–616.

[46] T. D. B. Le, F. Thung, and D. Lo, "Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools," *Empirical Softw. Eng.*, vol. 22, no. 4, pp. 2237–2279, 2017.

[47] X. Huo, F. Thung, M. Li, D. Lo, and S.-T. Shi, "Deep transfer bug localization," *IEEE Trans. Softw. Eng.*, vol. 47, no. 7, pp. 1368–1380, Jul. 2021.

[48] X. Sun, W. Zhou, B. Li, Z. Ni, and J. Lu, "Bug localization for version issues with defect patterns," *IEEE Access*, vol. 7, pp. 18811–18820, 2019.

[49] M. Pradel, V. Murali, R. Qian, M. Machalica, E. Meijer, and S. Chandra, "Scaffle: Bug localization on millions of files," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2020, pp. 225–236.

[50] G. Yang, K. Min, and B. Lee, "Applying deep learning algorithm to automatic bug localization and repair," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, Brno, Czech Republic, Mar. 2020, pp. 1634–1641.

[51] Z. Zhu, Y. Li, Y. Wang, Y. Wang, and H. Tong, "A deep multimodal model for bug localization," *Data Mining Knowl. Discovery*, vol. 35, no. 4, pp. 1369–1392, Apr. 2021.

[52] A. Qayum and A. Razzaq, "The impact of features on feature location," in *Proc. Int. Conf. Frontiers Inf. Technol. (FIT)*, Islamabad, Pakistan, Dec. 2019, pp. 1–15.

[53] A. A. Seyam, A. Hamdy, and M. S. Farhan, "Code complexity and version history for enhancing hybrid bug localization," *IEEE Access*, vol. 9, pp. 61101–61113, 2021.

[54] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Trans. Softw. Eng.*, vol. 46, no. 8, pp. 836–862, Aug. 2020.

[55] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, "Duplicate bug report detection using dual-channel convolutional neural networks," in *Proc. 28th Int. Conf. Program Comprehension*, Seoul, South Korea, Jul. 2020, pp. 117–127.

[56] K. E. E. Swe and H. M. Oo, "Source code retrieval for bug localization using bug report," in *Proc. IEEE 15th Int. Conf. Intell. Comput. Commun. Process. (ICCP)*, Cluj-Napoca, Romania, Sep. 2019, pp. 241–247.

[57] J. Zhang, R. Xie, W. Ye, Y. Zhang, and S. Zhang, "Exploiting code knowledge graph for bug localization via bi-directional attention," in *Proc. 28th Int. Conf. Program Comprehension*, Seoul, South Korea, Jul. 2020, pp. 219–229.

[58] T. Dilshener, M. Wermelinger, and Y. Yu, "Locating bugs without looking back," *Automated Softw. Eng.*, vol. 25, no. 3, pp. 383–434, Sep. 2018.

[59] A. R. Chen, T.-H.-P. Chen, and S. Wang, "Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs," *IEEE Trans. Softw. Eng.*, early access, Apr. 6, 2021, doi: 10.1109/TSE.2021.3071473.

[60] P. Jitngernmadan and K. Miesenberger, "A comparative study on Java technologies for focus and cursor handling in accessible dynamic interactions," *Stud. Health Technol. Inform.*, vol. 217, pp. 267–273, Jan. 2015.

[61] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? A two-phase recommendation model," *IEEE Trans. Softw. Eng.*, vol. 39, no. 11, pp. 1597–1610, Nov. 2013.

[62] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Silicon Valley, CA, USA, Nov. 2013, pp. 345–355.

[63] X. Ye, R. Bunescu, and C. Liu, "Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation," *IEEE Trans. Softw. Eng.*, vol. 42, no. 4, pp. 379–402, Apr. 2016.

[64] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Zurich, Switzerland, Jun. 2012, pp. 14–24.

[65] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of IR-based fault localization techniques," in *Proc. Int. Symp. Softw. Test. Anal.*, Baltimore, MD, USA, Jul. 2015, pp. 1–11.

[66] X. A. Harrison, J. D. Blount, R. Inger, D. R. Norris, and S. Bearhop, "Carry-over effects as drivers of fitness differences in animals," *J. Animal Ecol.*, vol. 80, no. 1, pp. 4–18, Jan. 2011.

[67] K. Suresh, "An overview of randomization techniques: An unbiased assessment of outcome in clinical research," *J. Hum. Reproductive Sci.*, vol. 4, no. 1, pp. 8–11, Jan. 2011.

[68] C. Y. Chong and S. P. Lee, "Analyzing maintainability and reliability of object-oriented software using weighted complex network," *J. Syst. Softw.*, vol. 110, pp. 28–53, Dec. 2015.

[69] Y. Zhu, Y. Huang, N. Jiang, and L. Chen, "A new model of software network for object-oriented software system," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Melbourne, VIC, Australia, Oct. 2021, pp. 516–522.

[70] H. Li, T. Wang, W. Pan, M. Wang, C. Chai, P. Chen, J. Wang, and J. Wang, "Mining key classes in Java projects by examining a very small number of classes: A complex network-based approach," *IEEE Access*, vol. 9, pp. 28076–28088, 2021.

ABDUL QAYUM received the bachelor's degree in information technology (IT) from the University of Sargodha, Gujranwala, Pakistan, in 2018. He is currently pursuing the master's degree with COMSATS University Islamabad (CUI), Islamabad, Pakistan. This publication is a part of his thesis. His research interests include automatic software maintenance, feature location, bug localization, information retrieval models, human-centric tools and techniques, and source code analysis. In addition, he has development experience in Java, Python, and mobile application development.

SAIF UR REHMAN KHAN received the bachelor's degree in computer science and the master's degree in software and system engineering from Mohammad Ali Jinnah University (MAJU), Islamabad, Pakistan, in 2005 and 2007, respectively, and the Ph.D. degree in software engineering from the University of Malaya (UM), Kuala Lumpur, Malaysia, in 2018. He has been with the Faculty of the Department of Computer Science, COMSATS University Islamabad (CUI), Islamabad, since 2005. He has more than 19 years experience of teaching, research, and development. He has published numerous research articles in high-impact journals and peer-reviewed conferences. His research interests in software engineering include verification and validation, search-based software engineering, cyber-physical systems, requirements engineering, and software project management. He has been in several expert review panels, both locally and internationally. He was a recipient of the Best Paper Presentation Award from UM, in 2014, and a Certificate of Outstanding Contribution in Reviewing from *FGCS* journal, in 2018.

INAYAT-UR-REHMAN received the Ph.D. degree in computer science (e-learning) from COMSATS University Islamabad (CUI), Islamabad, Pakistan, in 2017. He has over 18 years of teaching experience and is currently working as an Assistant Professor with the Department of Computer Science, CUI. He is extensively involved in conducting training for basic computing courses for national and multinational companies. In the e-learning domain, his research interests include computer-assisted core in education, designing learning tools, computer animations for learning, HCI for design in learning tools, cognitive learning, and the use of educational psychology for e-learning applications. His research interests in software engineering domain include software testing, software project management, and software reusability.

ADNAN AKHUNZADA (Senior Member, IEEE) is currently working as an Associate Professor with the Faculty of Computing and Informatics, Universiti Malaysia Sabah, Malaysia. His experience as an Educator and a Researcher is diverse that includes an Assistant Professor at COMSATS University Islamabad (CUI); a Senior Researcher at RISE SICs Vasteras AB, Sweden; a Research Fellow and the Scientific Lead at DTU Compute, Technical University of Denmark (DTU); the Course Director of ethical hacking at The Knowledge Hub Universities (TKH), Coventry University, U.K.; a Visiting Professor having mentorship of graduate students; and a Supervisor of academic and research and development projects both at UG and PG levels. He has a proven track record of high-impact published research and commercial products. He has also been involved in international accreditation, such as Accreditation Board for Engineering and Technology (ABET) and curriculum development according to the guidelines of ACM/IEEE. He is a PI of national and a Co-PI of several Swedish and Horizon 2020 EU funded projects. His research interests include cyber security, secure future internet, artificial intelligence, such as machine learning, deep learning, and reinforcement learning, large scale distributed systems, such as edge, fog, cloud, and SDNs, the IoT, industry 4.0, and the Internet of Everything (IoE). He is a member of technical program committee of varied reputable conferences, journals, and editorial boards. He is also a Professional Member of ACM with extensive 13 years of research and development (R&D) experience both in ICT industry and academia.