Using Fuzzy Code Search to Link Code Fragments in Discussions to Source Code

Nicolas Bettenburg, Stephen W. Thomas, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University
Kingston, Ontario, Canada
Email: {nicbet, sthomas, ahmed}@cs.queensu.ca

***Abstract*—When discussing software, practitioners often reference parts of the project's source code. Such references have different motivations, such as mentoring and guiding less experienced developers, pointing out code that needs changes, or proposing possible strategies for the implementation of future changes. The fact that particular parts of a source code are being discussed makes these parts of the software special. Knowing which code is being talked about the most can not only help practitioners to guide important software engineering and maintenance activities, but also act as a high-level documentation of development activities for managers. In this paper, we use clone-detection as specific instance of a code search based approach for establishing links between code fragments that are discussed by developers and the actual source code of a project. Through a case study on the Eclipse project we explore the traceability links established through this approach, both quantitatively and qualitatively, and compare fuzzy code search based traceability linking to classical approaches, in particular change log analysis and information retrieval. We demonstrate a sample application of code search based traceability links by visualizing those parts of the project that are most discussed in issue reports with a Treemap visualization. The results of our case study show that the traceability links established through fuzzy code search-based traceability linking are conceptually different than classical approaches based on change log analysis or information retrieval.**

I. Introduction

In "The Cathedral and the Bazaar" [26], Eric Raymond notes that one of the main advantages of open-source development is the reduced rate of software defects grounded in Linus' Law, i.e., "a direct result of the increased communication among developers about the source code". Understanding the impact of such developer communication on software quality has been the focus of recent research [4], [8] and is based on the explicit and implicit knowledge of developers that is recorded during the development of a software system.

Implicit developer knowledge is embedded in a variety of repositories, such as mailing list archives, modification requests, issue reports, the source code itself and accompanying documentation. Often, this implicit knowledge is of informal nature and consists of a mixture of natural language texts and structural elements that refer to the project's source code. Links between the source code and the surrounding documentation have been recognized in the past as an important factor for effective software development, and as a result, software engineering research spends much effort in uncovering such traceability links [24].

Past approaches to uncovering traceability links between documentation and source code are commonly based on information retrieval [2], [15], [18], [20], natural language processing [1], [19] and lightweight textual analyses [5], [11], [12]. Each approach, however, is tailored towards a specific set of goals and use cases. For example, when linking code changes to issue reports by analyzing transaction logs [28], we observe only the associations between a bug report and the final locations of the bug fix, but miss the bug fixing history: all the locations that a developer had to investigate and understand before he could find an appropriate way to fix the error.

In this paper, we aim to find traceability links between issue reports and source code. For this purpose, we propose a new approach that uses token-based clone detection as an implementation of fuzzy code search for discovering links between code fragments mentioned in project discussions and the location of these fragments in the source code body of a software system. In a case study on the `ECLIPSE` project, we first extract source code fragments from bug report discussions and then use the `CCFinder` clone detection tool, a readily available implementation of fuzzy code search, to identify all occurrences of the extracted code fragments in the software system's source code. We explore the value of the resulting traceability links through a quantitative evaluation and compare the resulting traceability links to those established by two classical approaches: change log analysis, which is the state-of-the-art for linking issue reports to source code, and information retrieval.

Our work makes the following contributions to the research area: first, we *establish a new class of traceability links* that link code fragments contained in project discussions to the actual occurrences of these fragments in the source code body of a software system. Second, we report on *a qualitative and quantitative analysis* of our approach. Third, we *demonstrate an example application* of this new class of traceability links through identification and visualization of those parts of the software system that are discussed the most. In the future, we envision traceability links established by our approach to be used to assist practitioners when browsing issue reports. A sample application would be an enhanced `BugZilla` system as illustrated in Figure 1, which assists the bug fixing process by identifying code fragments contained in the corresponding

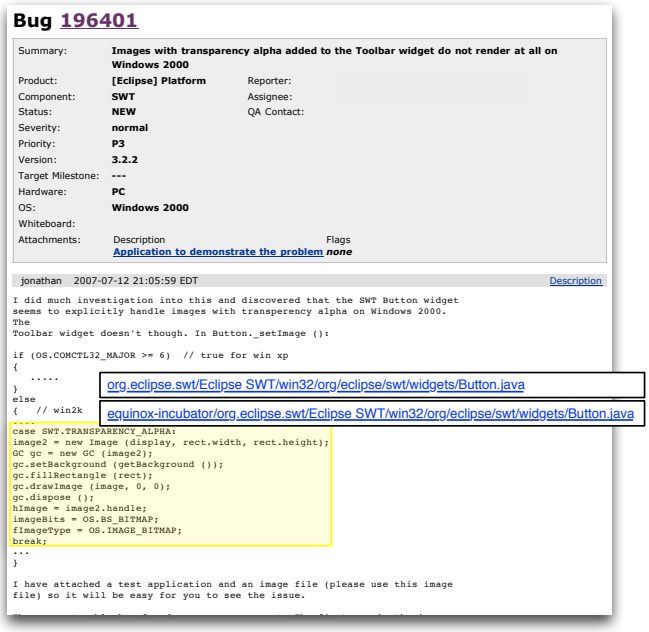

Fig. 1: Example of an enhanced bug report system that points developers to similar code.

issue report discussion and points developers to the locations in the project's source code that contain similar or identical code.

The rest of this paper is organized as follows: in Section 2, we present related work to our study, followed by a discussion of the limitation of existing approaches in the context of linking code fragments contained project discussions to source code. Together with this discussion, we present our fuzzy code search-based approach to traceability linking in Section 3. In Section 4 we present the design and results of a case study on the ECLIPSE software system, followed by a sample application of the obtained traceability links to uncover the parts of the software that are discussed the most (Section 5). We conclude our work in Section 6 with a discussion of our results and future research avenues.

II. Background and Related Work

Previous work in the area of traceability link recovery between source code and documentation can be categorized into three groups: information retrieval-based approaches, approaches that analyze the change logs of transactions to the version control system, and lightweight textual approaches that scan documents for code entities. In the following we summarize the research in each of these categories.

A. Information Retrieval Approaches

Frakes and Nejmeh pioneered the use of information retrieval approaches to support software reuse. Their CATALOG system implemented an interactive search engine for source code documents based on search terms supplied by the user [14].

Maarek et al. extended this idea in their work on automatically constructing software libraries with the help of information retrieval methods [19]. Attributing the moderate success of software reuse as a lack of a central library for locating and understanding code and documentation, the authors propose to use information retrieval methods to group sets of unorganized documents into software libraries, thus connecting source code with surrounding documentation.

Technique	Intended Purpose
Information Retrieval [2], [3], [20]	Establishing traceability links for software requirements.
Change Log Analysis [10], [12], [28]	Associating changes to the source code with issue reports.
Lightweight Analysis [5], [6]	Linking mailing list discussions to source code entities mentioned in the discussion.
Code Search (This work)	*Linking code fragments extracted from project discussions to their location in the project's source code.*

TABLE I: Overview of existing linking approaches.

Antoniol et al. recognized that the domain-specific knowledge of developers is implicitly encoded in such surrounding documentation in the form of mnemonics for identifiers that capture high-level program concepts [1].

In an extension to their former work, Antoniol et al. studied the use of vector-space information retrieval models for recovering traceability links between source code and free-text documentation [2]. In their case studies on two software systems, Antoniol et al. found that both approaches yield very high recall, with both approaches finding up to 100% of the existing links.

Marcus et al. extend Antoniol's idea by investigating the use of a novel vector-space information retrieval model for traceability link recovery [20]. In their work they demonstrate the suitability of Latent Semantic Indexing (LSI) to the domain of source code and documentation, while being computationally less expensive than the previous approaches by Antoniol et al.

Cubranic et al. link source code to documents, tasks, persons and messages in their HIPIKAT project memory system [11]. For a given artifact, they establish links to similar artifacts by calculating document similarities based on an LSI vector space model similar to Marcus et al. [20]. Through two case studies they show that the approach successfully provides pointers to files needed for the specific modification tasks.

In a study on automatic generation of traceability links between arbitrary software artifacts, De Lucia et al. extend a software artifact management system called ADAMS with a traceability recovery approach based on LSI [18]. Their case study revealed that information retrieval-based traceability link recovery suffers from a high number of false positives requiring much manual effort from users to discard incorrect links.

Information retrieval techniques, especially vector-space models have been demonstrated to be successful for automatically identifying semantic connections between source code and surrounding documentation. Jiang et al. were the first to note that previous techniques could not effectively and automatically deal with software evolution [15]. As a solution to the problem of changing documents over time, they proposed an incremental LSI technique and implemented the approach

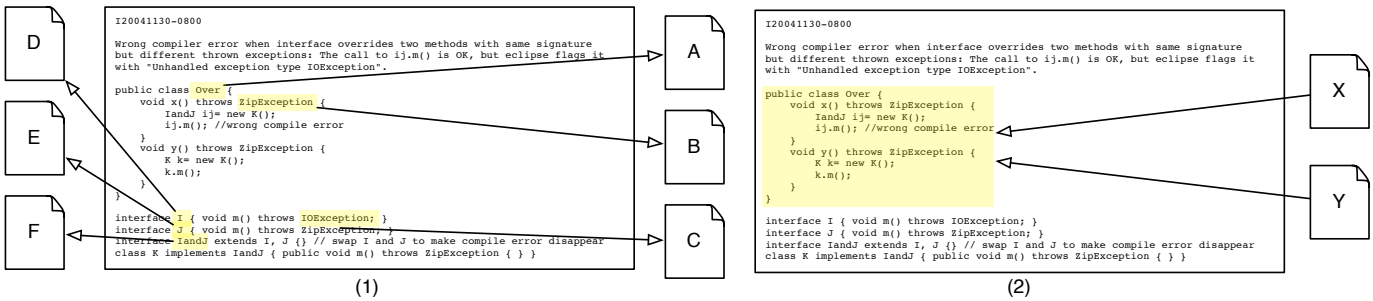

Fig. 2: Lightweight textual analysis finds traceability links from code entity names to files implementing those entities (1). Fuzzy code search links the entire code fragment to the actual occurrences of the fragment in the projects' source code (2).

in an automated traceability link evolution management tool.

Asuncion et al. presented another incremental IR approach, which uses latent Dirichlet allocation (LDA) for capturing traceability links [3]. They show in a case study that LDA performs as well or better than LSI with respect to precision and recall of the captured traceability links, while being computationally less expensive than LSI.

B. Change Log Analysis

In addition to information retrieval approaches, there exist a variety of specialized approaches for establishing traceability links. These approaches rely on implicit knowledge about software development repositories to establish traceability links between source code and documentation.

Two of the most prominent approaches, which links source code to bug reports, were introduced by Fischer et al. [12], [13] and Cubranic et al. [11], who discovered that developers tend to include bug report identifiers in the change logs of version control system transactions. This information can be used to link the files affected by a transaction to the bug report mentioned in the change log message.

Sliwerski et al. [28] refined this approach and introduced a set of heuristics that measure the syntactic and semantic relevance of links to cut down the number of false positive links reported by this method.

Associating source code and issue reports through the analysis of change log messages has been widely adopted in the defect prediction community, forming datasets that are at the core of many research efforts in the area [17], [22], [27].

However, Bird et al. note that these datasets suffer from inconsistent linking and represent only a smaller sample of links from the population of all possible links between issue reports and the source code [10].

C. Lightweight Textual Analysis

As information on bug report identifiers is limited to commit log messages, change log analysis cannot be used to link other forms of documentation to source code artifacts. Bacchelli et al. hence proposed a set of lightweight techniques [5] to establish traceability links between mailing list discussions and source code, based on the recognition of entity names, such as classes definitions or method names, inside the textual contents of messages and linking them to their corresponding implementation files.

D. Limitations of existing approaches

Table I summarizes existing approaches discussed so far. Each of these approaches has been designed to meet a specific goal and they have been shown to perform well for their intended use cases. In this paper, we want to consider the context of establishing traceability links between issue reports and source code files. In particular, we focus on establishing links between code fragments contained in bug report discussions to their occurrences in the project's source code. Within this intended use-case we identified the following limitations for the applicability of existing approaches:

Information retrieval (IR) methods, such as Latent Semantic Indexing (LSI), latent Dirichlet allocation (LDA), or Vector-Space Models (VSM) are designed to identify commonly occurring concepts and patterns across combined collections of source code documents and surrounding documentation. However, IR techniques rely on a user-defined number of dimensions that limits the amount of concepts derived and thus the specificity of uncovered traceability links. Additionally, IR techniques usually rely on a repetition of text features in order for them to emerge as concepts, whereas code fragments are often small (compared to the size of whole files of source code), and often lack the required repetition of features that is required in this approach.

Change log analysis based traceability linking requires that an issue has been filed through the bug report system, and that the issue has to have led to an actual change of the source code. Additionally, when fixing bugs, developers commonly discuss different implementations and often discuss many parts of the source code with the final fix possibly taking place in a completely different part of the source code. Links established through change log analysis record the final location of a much more involved and complex process and ignore the history of the bug fixing process.

Lightweight textual analysis approaches, for example as presented by Bacchelli et al. [5], focus on linking names of source code entities mentioned in developer discussions, such as identifiers and types, to the implementation files of these entities. These links are conceptually different from our approach, as we do not want to link entity names to

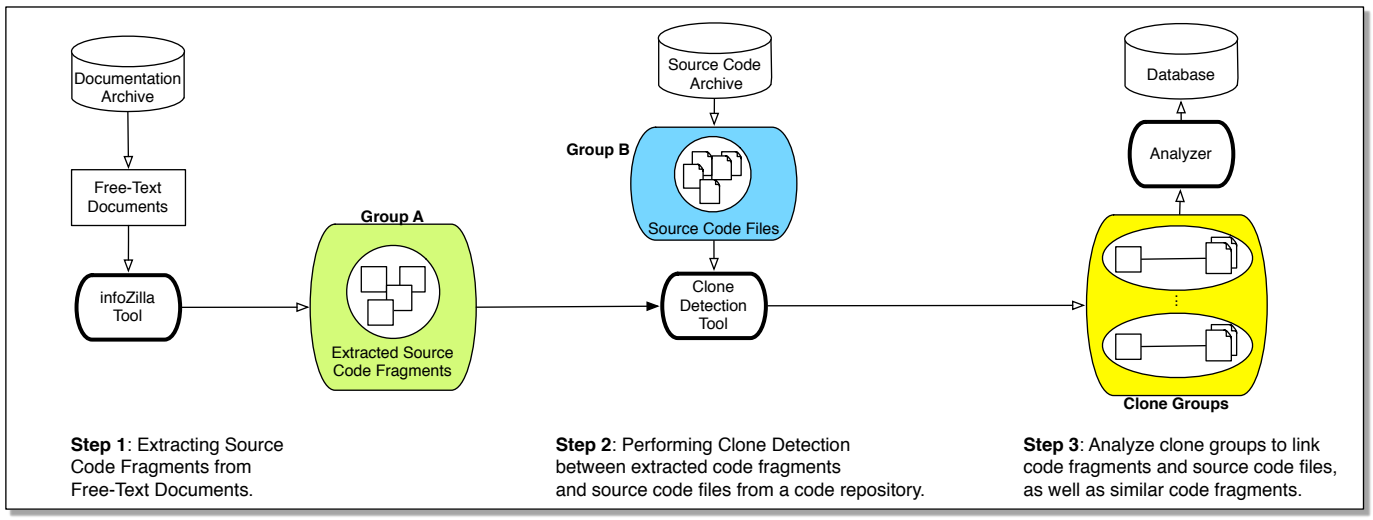

Fig. 3: Overview of a clone-detection based approach to locate code fragments in a project's code base.

the implementation of these entities, but whole fragments of source code examples to the locations in a projects source code where these fragments occur. To illustrate the conceptual difference of the links created by this approach, consider the bug report shown in Figure 2. In this example, lightweight textual analysis would recognize the types `Over`, `ZipException`, `IOException`, and interface types `I`, `J`, and `IANDJ`. For each recognized type, a traceability link is established to those source code files `A` to `F` that define and implement these types (e.g., `some/path/ZipException.java`).

In fuzzy code search-based traceability linking, the complete code fragment contained in the discussion would be recognized as a smaller subset of code that is contained in some source code files `X` and `Y` (e.g., `some/path/TestCase1.java`).

III. Code Search Based Traceability Linking

In order to locate source code fragments contained in project discussions within the source code body of a software system, we propose to use clone-detection as a readily available fuzzy code search implementation. An general overview of our proposed approach is illustrated in Figure 3.

Step 1: First, we use the `infoZilla` tool [9] to extract source code from documents obtained from a documentation archive (Step 1). The `infoZilla` tool uses a combination of regular expressions and island parsing [21] to identify and extract source code regions from free-form text documents with high reliability. At first, the complete textual input is treated as "water". Using regular expressions it identifies common program elements, such as assignments, method calls or loops. These elements form "islands" in the water. The tool then examines the text surrounding each island to determine whether this text is code and grows the island accordingly. For a more detailed discussion of the tool we refer to our previous work [7]. Each extracted code region is stored in a separate file, uniquely identifying the original discussion document from which the code was extracted. We refer to the complete collection of extracted code regions as "Group A".

Step 2: In order to discover where each source code fragment of Group A appears in the source code of the project (we refer to the collection of source code files of a project as "Group B"), we use a token-based clone detection tool to carry out a fuzzy textual search. We call this *code search*. We stress that the fuzzy search aspect is critical: during the manual inspections of source code and code fragments carried out during design and refinement phase of our work, we found that in practice, occurrences of code fragments in the actual source code body of the project are often slightly modified versions (e.g., by normal evolution of the source code, or to adapt code examples to particular APIs) of the original fragment contained within discussions.

Step 3: Clone detection tools usually report their findings in terms of clone pairs and clone groups. A clone pair associates a source code region in a file f_1 with a corresponding duplicated code region in another file f_2. All clone pairs with identical duplicated code regions are grouped together and form a clone group. We analyze all clone groups for clone pairs that associate files with extracted code fragments (Group A) to source code files (Group B). A traceability link as established by our proposed approach is hence a tuple containing a unique clone group identifier, a file path that corresponds to a code fragment, a file path that corresponds to a source code file, and a description of the exact location of where the code fragment occurs within the source code file. We store all traceability links in a database for further analysis.

IV. Case Study

In this section, we present a case study on the `ECLIPSE` software system. We apply our proposed approach to discover traceability links between discussions attached to issue reports

contained in the projects BugZilla bug tracking system, and the complete source code of the software contained in the project's CVS software archive.

We first perform a quantitative evaluation that illustrates the performance of our proposed approach with respect to the amount and validity of traceability links established. We then proceed with a qualitative analysis that first compares traceability links established through fuzzy code search to traceability links established through change log analysis, the state of the art method to link issue reports to source code files contained in a version control system. We finish our analysis with a discussion of the use of information retrieval based traceability linking and its shortcomings in the presented use case.

A. Data Collection

Based on past interest in establishing links between developer discussions and code [6], as well as the significance of links between issue reports and source code for defect prediction [10], we chose to use the descriptions and discussions attached to issue reports of ECLIPSE as our main source of project discussions. We followed the approach by Zimmermann et al. [32], and extracted a total of 211,843 issues from the BugZilla issue tracking system of the ECLIPSE project, that were filed from ECLIPSE version 2.0 until ECLIPSE version 3.2. Additionally, we obtained a snapshot of the complete software archive of the ECLIPSE 3.2 release. This snapshot contains both, the project's source code, and a complete record on the history of all changes to the source code that were carried out by developers.

In order to perform clone detection, we use one of the most popular token-based clone detection tools, CCFinder [16]. This choice is mainly motivated by the high recall of token-based clone-detection, paired with its good scalability. Furthermore, token-based clone detection approaches have the advantage over other approaches that they can work with uncompilable code, such as commonly found in code fragments of discussions.

We adapted the CCFinder tool, which was originally written for the Windows platform, to a 64 bit version of Ubuntu Linux Server 9.10. These modifications allowed us to perform our experiments in main memory, greatly increasing performance and allowing us to work on the complete copy of the project's source code. Using the CCFinder tool, we executed an inter-group clone-detection between the code extracted from issue reports (Group A in Figure 3) and the project's source code from the version control system (Group B in Figure 3). Code clones are reported by CCFinder as a set of clone groups.

Each clone group is associated with a unique identification number and contains a sequence of clone pairs that describe the exact locations of a code clone between two files f_1 and f_2. As we ran the clone detection tool with the option to carry out an inter-group analysis, each clone pair reports on the occurrence of a cloned instance of code from a file belonging to group A in a file from group B.

Unfortunately, CCFinder reports one pair for each possible permutation of pairs that can be obtained from the set of clones in a clone group. In order to prune this representation, we transform the results reported by CCFinder in the analysis step and identify unique triples of clone group identifier, absolute filename, and the exact cloned code. These triples are then stored in a database for further analysis.

Using the infoZilla tool, we extracted code fragments from the discussions of all 211,843 ECLIPSE issue reports. A total of 33,301 issue reports contained source code fragments, and of these, a total of 17,748 reports contain code fragments with a length of more than 30 source code tokens, which is the minimum amount of tokens needed for clone detection by CCFinder. From those 17,748 issue reports, we removed another 10,042 instances that CCFinder failed to transform into a token stream. The transformation requires a certain amount of context, such as basic blocks, for inferring token types, and code fragments in these instances did not include enough context for CCFinder's transformation. This leaves us with a total of 5,511 issue reports that form the base of our analysis.

B. Quantitative Analysis

Overall, our approach was able to establish a total of 47,783 traceability links, which connect 3,865 out of 5,511 (70.13%) of the issue reports to a total of 13,581 out of 51,600 (26.32%) ECLIPSE source code files. We found that on average, each issue report (which might contain multiple different code fragments) was linked to 5.67 source code files.

Our approach failed to correctly process 1,646 of the 5,511 (29,87%) issue reports. In order to better understand, why these issue reports could not be linked to the project's source code, we performed a manual investigation. Overall, we identified the following two main causes:

1) **Unrelated code.** Many discussions contain code fragments that are not part of the project's source code body. For example, bug #99986 describes a problem with ECLIPSE's handling of inheritance in a specific build. However, the code example used to demonstrate the problem was not (yet) part of the ECLIPSE 3.2 source code.

2) **Code evolution.** Source code, especially code discussed in issue reports, often undergoes significant changes during the evolution of a system. For our experiment, we used a single snapshot of the software system's source code (version 3.2) within which we search for occurrences of the extracted code fragments. However, code fragments extracted from discussions might have undergone significant changes, especially in the case of discussions that refer to much earlier versions of the source code, up to a point that even fuzzy code search failed to relate the extracted code fragments to any part of the project's source code. One possible solution to this problem would be to take an evolutionary approach

and also consider past snapshots of the software system source code, which are closer to the date of the discussion from which code fragments were extracted. We leave the investigation of this solution to future work.

C. Qualitative Analysis

To perform a qualitative analysis of our fuzzy code search-based approach, we first compare the traceability links established by our approach to traceability links established by the most prominently used approach in defect prediction: change log analysis. For this purpose, we implemented a change log parser, closely following the algorithm proposed by Sliwerski et al. [28], as well as taking the enhanced heuristics described by Bird et al. [10] into account. We applied this parser to the complete change history of the `ECLIPSE` version control repository and recorded all associations between issue reports and source code files.

Overall, the change log-based linking approach was able to link 16,722 issue reports to 23,079 source code files, with an average of 4.57 linked files per bug report. Of these 16,722 reports, a subset of 2,980 issue reports contain code fragments ($L \cap C$ highlighted in Figure 4a) and a subset of 507 reports were also linked by our approach ($O \cap C$ highlighted in Figure 4b). Taking the union of links established by both approaches, would result in a 20.01% increase in linked issue reports.

The very small intersection of the sets of issue reports linked by both approaches is notable: fuzzy code search-based linking creates many links between issue reports and source code that are not found by a change-log based approach and vice versa.

To explore the differences between both linking approaches, we randomly picked a sample of 10 reports each from the set of issue reports linked by both approaches ($O \cap L$), the set of issue reports linked by our approach but not by change log analysis ($O - L$), and the set of issue reports linked by change log analysis but not our approach ($L - O$). For each case in the random sample, we explore the established traceability links in the background of the corresponding issue reports and the discussion attached to them. We present our findings below.

1) Traceability links found by both approaches

The discussion of bug #31670, contains a sample class to illustrate a problem with the debugger. The source code used in the illustrative example creates a test case in the project's test suite. In addition to the original bug described, we learn that the corresponding transaction fixed another bug that was found during the fixing process. Traceability links of our approach point to the test case created for the original issue, whereas the traceability links of the commit log additionally point to the fix locations of both bugs.

In the discussion of bug #34593, a developer suggests a possible fix for the reported problem. Traceability links by both approaches point to the actual location of the fix that was carried out later on.

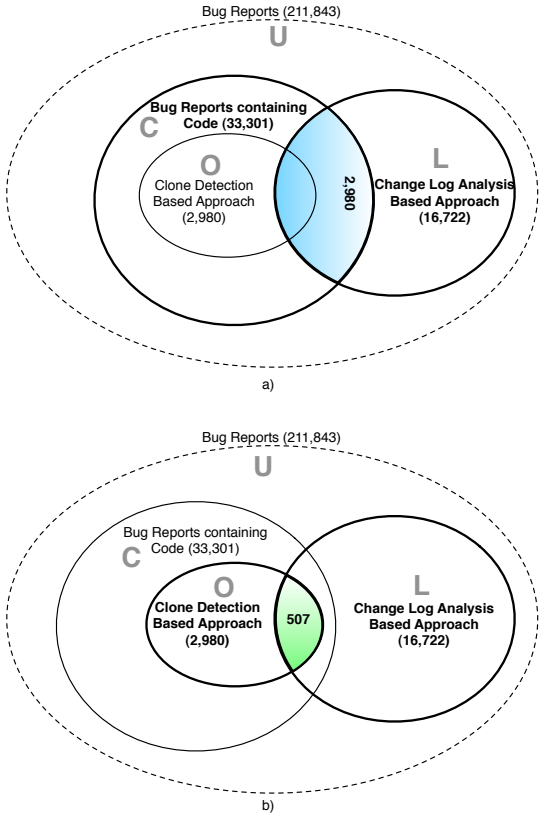

Fig. 4: Venn diagrams of the set of issue reports linked through our clone detection based and a change log analysis based approach.

In the discussion of bug #51821, a developer proposes a sample patch to address the described issue. Based on the source code contained in the proposed fix, our approach links to the same location in the project's source code, that was later modified to fix the issue – the proposed patch was actually applied.

In the discussion of bug #61605, a developer proposes a new try-finally paradigm to enhance the robustness of a plugin. The proposed method is welcomed by peers and applied to plugins throughout the project. Traceability links of both approaches capture all modified files.

As reported in the discussion of bug report #87288, the reported issue and corresponding bug fix had a "large impact on downstream components". We can observe this impact through the traceability links established through our approach: 69 files contain code that is similar to the discussed code fragment. Of these, we observe 67 files that were changed in the bug fixing transaction.

During the course of fixing bug report #92017, developers added support for very large images in main memory. In addition to a link to the fix location established through change log analysis, our approach establishes an additional link to the implementation of drag and drop support for images that uses the same approach as proposed in the bug fix.

In each case of bugs discussed in reports #93208, #93577, #105356 and #195763 the actual bug fix is applied in the

source code locations linked through the discussed code examples. In all cases both approaches established the same traceability links.

In the case of bug #164939, a developers reports on a code fragment that he believes responsible for the issue. Based on the code fragment, our approach links to the location of the actual fix and an additional source code files that contains the same (erroneous) code, and which be believe should have been change accordingly.

2) Traceability links found by code search, but not found by change log analysis

In the discussion of bug #21273, a developer provides an extract of the code he believes to be responsible for the filed issue. Traceability links established by our approach through this code fragment point to the actual location of a later fix. Change log analysis cannot establish links, as the transaction log does not contain references to any bug identifier.

In the discussion of report #45945, a developer reports a code example to illustrate the experienced problem. The bug report is later closed without a fix due to an operating platform incompatibility and no files are modified. Our approach however, establishes a link to a source code file containing code that is similar to the illustrative example, missing the correct link.

For report #62224, heuristics of the change log analysis based approach fail to identify the bug identifier. Our approach is able to establish a link to one of the two fix locations through the code fragment contained in the attached discussion.

In a similar way, change log analysis is not able to find the bug identifier in the change log of bug #65729. Our approach establishes links to all fix locations based on the code fragment mentioned in the bug report's discussion.

In the discussion of report #71047, a developer posts an example code to demonstrate that he cannot reproduce the reported problem. The bug report is closed as WORKSFORME and no transactions take place. Traceability links established by our approach point to source code in the project that is similar to the developer's code example.

Through the code fragments discussed in reports #93467 #103266 and #106736, our approach establishes links to the exact fix locations. In all cases, change log analysis is unable to find these links, as no change logs ever contain the corresponding this bug identifiers.

Report #105447 is marked as a duplicate of report #137621 since a fix to #137621 reportedly fixes #105447 as a side effect. Even though both approaches find the same traceability links pointing to the fix location, they are associated with different issue reports.

Report #174125 is never fixed due to an operating system specific problem. As a result, no transactions take place that could be linked through change log analysis. However, the traceability links established by our approach point to source code that is similar to the presented code example.

We found that code extracted from issue reports #171912 and #171909 was linked to the same source code files and consequently grouped together in the results presented by the clone detection tool. Upon inspection of both issue reports we found that they are closely related and describe an issue that originates from the same parts of the project's source code.

3) Traceability links found by change log analysis, but not found by our approach

In the cases of reports #31573, #73908, #92579, #191862 and #202382, the code fragments contained in the discussions of these issue reports are used as illustrative examples. For example, the code in #92579 is intended to be visualized in the project outline view. In all cases, the code fragments are unrelated to the corresponding fixes and are not contained in the source code. Change log analysis is however able to establish traceability links to the final fix locations.

In the cases of reports #80455 and #182006, the code fragments contained in the discussions are below the minimum token length threshold and thus ignored by our approach. However, change log analysis is able to establish links to the fix locations of both bugs.

The code fragment extracted from the discussion of report #45468 consists exclusively of `javadoc`-style comments. `CCFinder` however, ignores code comments during clone detection and hence our approach cannot establish traceability links.

During the discussion of report #153932, a developer proposes code for a potential fix to the reported issue, but peer developers decide to modify a completely different part of the source code to address the problem. Our approach is unable to link the code of the proposed fix to any source code file.

D. Using Information Retrieval for Traceability Linking

In addition to comparing code search based traceability linking to change-log analysis, the state-of-the-art method for linking issue reports to source code, we want to compare our proposed approach to traceability links obtained through information retrieval models. In particular, we use the Vector-Space Model (VSM) latent Dirichlet allocation, following similar approaches presented in the literature [2], [3], as both approaches have been demonstrated valuable for establishing traceability links between source code and surrounding documentation.

A key finding in our previous work, and also very recently in related work, is that VSM is actually better than LDA for finding traceability links between source code and other related free-text documents [25]. VSM is more accurate (better precision), but LDA has better recall.

To perform traceability linking using LDA and VSM, we prepare our data following the standard approaches in the field: every extracted code fragment (Group A) and every source code file (Group B) is preprocessed by splitting identifiers, removing common English stopwords and stemming. We then use the combination of all documents in Group A and Group B to train an index. In the case of VSM, we weight each term by computing its term-frequency and inverse-document frequency (tf-idf).

The two IR models return a set of potential links between a given code fragment and source code documents, ordered by their similarity score (i.e., cosine distance for VSM and conditional probability [30] for LDA) in the model. In our case, each code fragment is potentially linked to hundreds or thousands of source code files, depending on if the two share common words or topics.

To determine the quality of the established links, we select a random sample for manual inspection. Sampling theory tells us that we need to inspect 64 samples to have a 10% margin of error and a 95% confidence interval. During our manual inspection, we inspected the top three links (by similarity score) for each code fragment. We found that, surprisingly, for each our 64 sampled code fragments, none of the top three links were accurate: the given code fragment was not found in the linked source code file. To illustrate why, consider the following code fragment extracted from ECLIPSE issue report #155726:

```
EObject eObject = (EObject)resource
                  .getContents().get(0);
```

The index models of both LDA and VSM would represent this code fragment by its four preprocessed terms `eobject`, `object`, `resourc`, and `content`. These terms are so general that the IR models will return thousands of possible matches, since thousands of source code files contain at least one of these terms.

This poor performance is a consequence of several assumptions that IR models make. First, IR models are based on the "bag of words" model, meaning that the order of terms in each document is ignored. When searching for exact code fragments, this is an obvious disadvantage. Second, the preprocessing steps (especially splitting and stemming) used by IR models result in very general terms that are contained in many documents, as we saw in the example above. Third, the common similarity measures (e.g., cosine distance and conditional probability) are too general for our specific application, as they reward *any* shared words or topics, which is not restrictive enough to eliminate false positive matches.

As a result, we conclude that IR based traceability linking is not suitable to reliably link code fragments contained in issue report discussions to occurrences of these fragments in the project's source code.

V. Which parts of the software system are discussed the most?

Previous research has noted the importance of traceability links for developers for software maintenance and source code comprehension [1], as they aid practitioners in creating mental models of the source code and providing hints to the location of specific code. We pick up this idea to demonstrate a sample application of the traceability links established by our proposed approach, as a natural extension to the association of code contained in project discussions to their occurrences in the project's source code.

Traceability links established through our approach are bi-directional: given a code fragment we can determine its location in the source code, but at the same same, given a location in the source code, we know in which discussion this source code was talked about. Through this association we can count the number of discussions that refer to a source code location.

Figure 5 presents a visualization of the most discussed components of the ECLIPSE software system. This visualization is inspired by Wattenbergs "Visualizing the Stock Market" [29] and summarizes data from the beginning of the project until version 3.2. To increase visibility, we grouped source code locations at directory level. Every box represents a source code directory of the ECLIPSE software system. Boxes are coloured according to how much source code in each directory has been discussed in issue reports. Completely black boxes denote directories that contain source code that is discussed in less than 10 issue reports, and the whiter a box, the more issue reports discuss the source code in this directory.

We can use this visualization to locate "discussion hotspots": they appear as bright coloured boxes (the brighter, the more discussed). Among the most discussed components of the software system are the graphical user interface, compiler, data binding and internal components such as the debugger. Surprisingly, our visualization of "discussion hotspots" suggests that a substantial amount of discussion takes place on source code contained in the `examples` directory of the `SWT framework`. In order to empirically validate this "hotspot", we inspected a random sample of traceability links that point to files in this subsystem. Notably, we observed that developers in ECLIPSE appear to extensively borrow from code fragments contained in issue reports for use in regression testing. These examples are saved in the form of code snippets, minimal stand-alone programs that demonstrate functionality such as API usage on the one hand, and act as a basis for different test cases on the other hand. Through the projects website[1], developers actively encourage users to contribute snippets through the `BugZilla` issue tracking system. Based on these observations, we conjecture that developers thus acknowledge the importance of code fragments contained in project discussions, and actively migrate code fragments into code snippets to be used for regression testing purposes.

VI. Conclusions

Similar to previous approaches, fuzzy code search-based traceability linking has a specific intended use case, and limitations within other use cases. The main focus of our approach lies on linking issue reports to source code, with the aim of *locating code* that is talked about in project discussions within the body of source code of the software system. As the source code body of large software systems can easily contain thousands of files, establishing such traceability links is no trivial task.

Our proposed solution leverages an existing token-based clone detection tool, `CCFinder`, which was designed to

[1] http://www.eclipse.org/swt/snippets/, last accessed April 2011

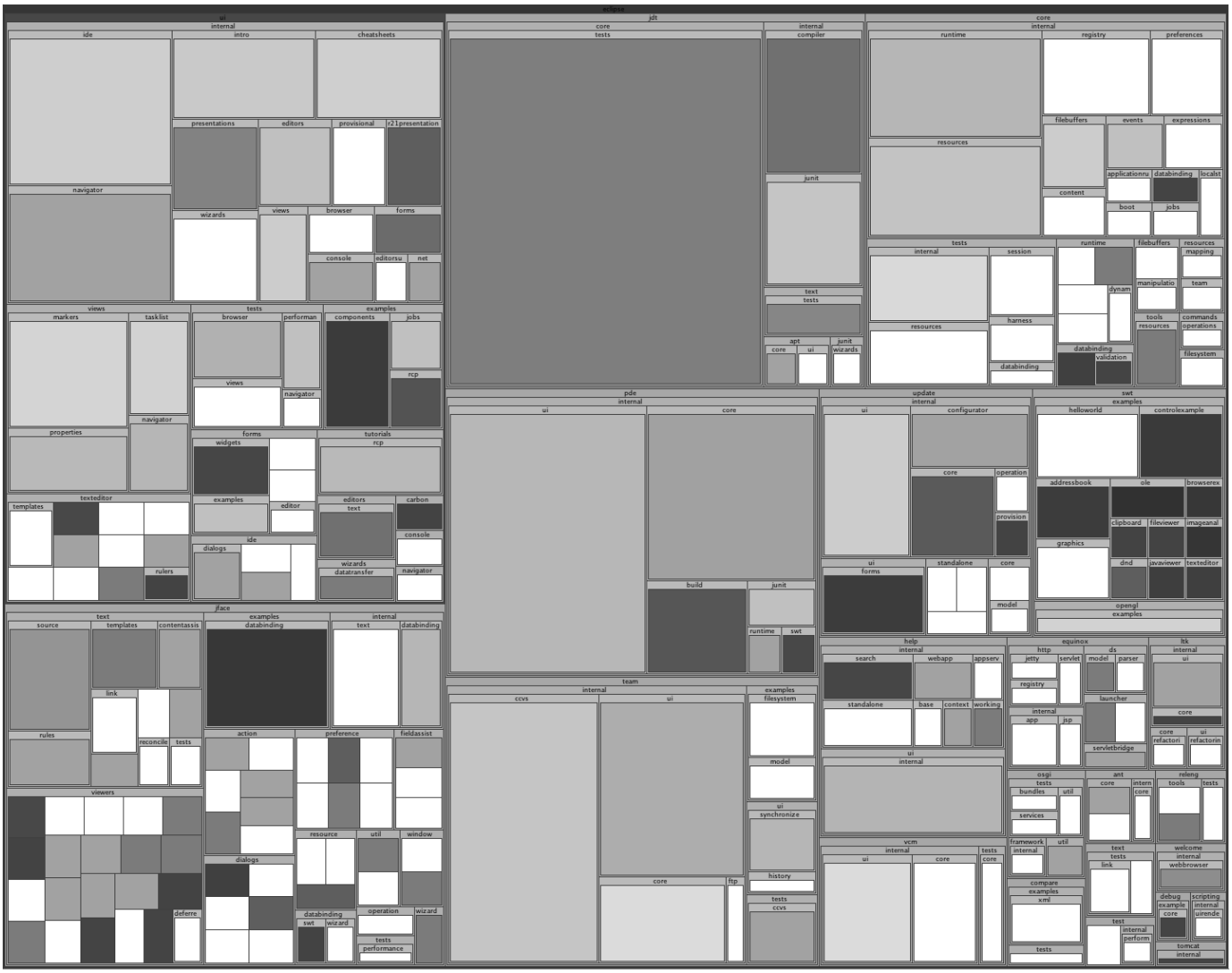

Fig. 5: Visualization of the most-referenced source code in Eclipse issue reports. Brightness indicates the amount of references and ranges from dark (rarely referenced) to bright (referenced often).

efficiently locate all the occurrences of similar code fragments in a software system, as a readily available implementation of fuzzy code search.

Through a case study on the ECLIPSE software project, we discovered that fuzzy code search-based traceability linking shares only a small percentage of traceability links with the state-of-the-art approach to link issue reports to source code files: commit-log analysis. We find that a combination of the sets created by both approach results in a 20.01% increased in total traceability links between issue reports and source code. We thus see a potential application of fuzzy code search-based traceability linking for recovering missing links, in the same vein of work presented by Wu et al. [31].

We demonstrated an example application of our approach: identifying and visualizing the parts of the software system that are discussed the most in issue reports. During the analysis of this visualization, we discovered that developers extensively borrow code fragments from project discussions for use in regression testing. Additionally, we identified a variety of interesting side effects of our traceability linking approach. For instance, we observed many clone groups that contain code fragments from multiple different discussions, i.e., traceability links established by our approach do not only link projects discussions to the source code, but also discussions of different issue reports among each other. As we have seen in section IV-C, our approach discovers links from multiple issue reports to the same source code files. Within this context, we see a potential application of our traceability linking approach for the identification of related, or duplicate issue reports – a research avenue that we plan to explore in future work.

The objective of our proposed fuzzy code search-based approach is to find high quality traceability links between issue reports and source code. Overall, we have found that among the three possibilities for establishing such links, only information retrieval models fail this objective, due to their assumptions and generality. As a result of our analyses, we

conclude that practitioners should use both change log analysis and fuzzy code search, but not information retrieval models. However, both the quantitative and qualitative analysis of our approach suggest that there is much room for improvement: about one third of issue reports needed to be discarded as a result of the limitations of the used clone detection tool. Hence, one major direction of our future work is to study the applicability of other fuzzy code search approach, for example approximate string matching techniques [23], to find occurrences of extracted code fragments in the project's code base.

REFERENCES

[1] G. Antoniol, G. Canfora, A. de Lucia, G. Casazza, and E. Merlo, "Tracing object-oriented code into functional requirements," in *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2000, p. 79.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, 2002.

[3] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *ICSE'10: Proceedings of the 32nd International Conference on Software Engineering*. IEEE Computer Society, 2010, p. to appear.

[4] A. Bacchelli, M. d'Ambros, and M. Lanza, "Are popular classes more defect prone?" in *Proceedings of FASE 2010: 13th Conference on Fundamental Approaches to Software Engineering*. Springer LNCS, 2010, pp. 59–73.

[5] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes, "Benchmarking lightweight techniques to link e-mails and source code," in *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 205–214.

[6] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *ICSE'10: Proceedings of the 32nd International Conference on Software Engineering*. IEEE Computer Society, 2010, p. to appear.

[7] N. Bettenburg, "Duplicate bug reports considered harmful?" Master's thesis, Saarland University, Facutly of Natural Sciences and Technology, Saarbruecken, Germany, July 2008.

[8] N. Bettenburg and A. E. Hassan, "Studying the impact of social structures on software quality," in *ICPC'10: Proceedings of the 18th IEEE International Conference on Program Comprehension*. IEEE Computer Society, 2010, p. to appear.

[9] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*. New York, NY, USA: ACM, 2008, pp. 27–30.

[10] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009, pp. 121–130.

[11] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 446–465, 2005.

[12] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, p. 90.

[13] ——, "Populating a release history database from version control and bug tracking systems," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003, p. 23.

[14] W. B. Frakes and B. A. Nejmeh, "Software reuse through information retrieval," *SIGIR Forum*, vol. 21, no. 1-2, pp. 30–36, 1987.

[15] H.-Y. Jiang, T. N. Nguyen, I.-X. Chen, H. Jaygarl, and C. K. Chang, "Incremental latent semantic indexing for automatic traceability link evolution management," in *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 59–68.

[16] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[17] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2006, pp. 35–45.

[18] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, p. 13, 2007.

[19] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Trans. Softw. Eng.*, vol. 17, no. 8, pp. 800–813, 1991.

[20] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135.

[21] L. Moonen, "Generating robust parsers using island grammars," in *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 13.

[22] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 181–190.

[23] G. Navarro and R. Baeza-yates, "Very fast and simple approximate string matching," in *Information Processing Letters*, 1999, pp. 65–70.

[24] R. Oliveto, G. Antoniol, A. Marcus, and J. H. Hayes, "Software artefact traceability: the never-ending challenge," in *ICSM 2007: Proceeedings of the 23rd IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 485–488.

[25] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceeding of the 8th working conference on Mining software repositories*, ser. MSR '11. ACM, 2011, pp. 43–52.

[26] E. S. Raymond, *The Cathedral and the Bazaar*, T. O'Reilly, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999.

[27] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. New York, NY, USA: ACM, 2006, pp. 18–27.

[28] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[29] M. Wattenberg, "Visualizing the stock market," in *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*. New York, NY, USA: ACM, 1999, pp. 188–189.

[30] X. Wei and W. B. Croft, "LDA-based document models for ad-hoc retrieval," in *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2006, pp. 178–185.

[31] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 15–25.

[32] T. Zimmermann, N. Nagappan, and A. Zeller, *Predicting Bugs from History*. Springer, February 2008, ch. Predicting Bugs from History, pp. 69–88.