Capturing Software Traceability Links from Developers' Eye Gazes

Braden Walters*,Timothy Shaffer[†]
*Department of Computer Science and Information Systems
[†]Department of Mathematics
Youngstown State University
Youngstown, Ohio 44555 USA
{bmwalters01, trshaffer}@student.ysu.edu

Bonita Sharif
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio 44555 USA
bsharif@ysu.edu

Huzefa Kagdi
Department of Electrical Engineering and Computer Science
Wichita State University
Wichita, Kansas 67260 USA
kagdi@cs.wichita.edu

ABSTRACT

The paper presents a novel approach for recovering software traceability links from developers' eye gazes. An eye tracker is used to capture eye gazes while developers perform software maintenance tasks within the Eclipse IDE. An algorithm is presented that establishes a set of traceability links from the eye-gaze data of several developer sessions. A preliminary study assesses the feasibility and validity of the approach. The links generated by the approach were validated by another set of developers. Results indicate that our algorithm achieves strong recall when developers accurately perform bug-localization tasks.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; H.1.2 [**User/Machine Systems**]: Human Factors

General Terms

Documentation, Human Factors

Keywords

eye-tracking study, traceability link recovery

1. INTRODUCTION

A software traceability environment/tool seeks to discover and maintain (evolve) links among artifacts as the software evolves over time. Traceability benefits are projected in several key software development and evolution tasks such as program comprehension, verification and validation, impact analysis, and testing. Although, the support for software traceability is strongly rooted in research convictions, the dichotomy is that software traceability is very primitive or often missing in practice [5]. One of the main reasons cited for this problem is the high error-prone human effort and costs required to document and maintain the traceability links over time [6]. Also, state of the art traceability support [2] suffers from a high number of false positives (indicating traceability when none exists); thereby burdening developers to manually examine and verify the candidate links for accuracy [4]. The forward engineering support for traceability imposes an additional burden of establishing traceability links on the developers, who are often faced with demanding delivery schedules in industry. Therefore, traceability often downgrades to a low-priority activity with no obvious immediate benefits to the developers.

The issue of human effort in traceability is of paramount importance and needs to be thoroughly addressed for software traceability to be successfully adopted in industry and to become a common developer routine. The Center of Excellence for Software Traceability (http://www.coest.org) has an alliance of investigators that identified several grand challenges [3] to be undertaken in order for software traceability to become more prevalent and sustainable in software development. One of these challenges is that traceability needs to be effortless.

In this paper, we present an algorithm that automatically determines links between a change request (e.g., a textual bug description) and source code entities from a set of developer eye tracking sessions. This approach to traceability recovery is developer centric and not artifact oriented. In that, we observe the developer's work in an unobtrusive manner with eye-tracking equipment. The traditional approaches have relied mostly on analyzing the content of artifacts. We conduct a comparison to true links and a validity check with another set of developers to determine if the links retrieved were actually useful to them. Our preliminary results are promising. In addition, developers that validated links stated that the links uncovered by our algorithm would assist them in the task.

The goals of this new and emerging idea are two-fold. First, we show that it is feasible to use eye gaze as input to retrieving traceability links. That is, what developers look at in their IDE is a sound source for traceability links, without incurring too much noise, in a realistic scenario. Second, we show that the links retrieved are potentially useful to developers and that recall is high. To meet these two goals, a pilot study was conducted using *iTrace*, our Eclipse plug-in that implicitly collects gaze data while developers work.

2. THE APPROACH

An overview of our approach is presented in Figure 1. The main working environment is the *iTrace* Eclipse plug-in. After the source code of the subject system (iTrust in this case) is loaded into Eclipse and the developer is ready for a session, *iTrace* is able to generate XML gaze data after connecting to the eye tracker. The eye tracker gives us raw data in the form of (x,y) coordinates of where the developer is looking. *iTrace* maps these coordinates to line and column numbers for all artifacts the developer looks at in Eclipse, to accurately give us fine-grained line-level granularity.

Next, the *XML Gaze Data* is fed into the *GazeReader* module. The *SimpleGraph Gaze-Link* algorithm is then run on the data. The results of the algorithm are task-to-source entity links and an optional graph of linked entities. The shaded area in the diagram denotes the offline post processing done on the *XML Gaze data*. As part of future work, the processing will be incorporated into *iTrace* where the links would be generated immediately after the session for the developer to validate.

3. THE ALGORITHM

The *SimpleGraph Gaze-Link* algorithm finds links between individual source code entities (SCEs) and a task. Implicitly, any links discovered between a task and SCEs will generate links between those same SCEs and the task's related documentation (e.g. use cases). The algorithm begins by getting source code information from srcML (http://www.srcML.org) and parsing each developer's gaze data for the current task. The algorithm is described below.

For each subject, a weighted digraph is generated with SCEs (which may be methods or member attributes) as vertices and the gaze path that the developer took between SCEs represented as edges between these vertices. The first time an edge is encountered while scanning the gaze data chronologically, its weight is set to the square of the timestamp (which is recorded by the eye tracker) of that gaze. Using the square of the timestamp is simply a way to give higher weight to new links found later in the session. For each subsequent time that the edge is encountered, its timestamp is added to the current weight. G_s is the set of SCE link weights for dataset s, i.e., when gaze jumps from SCE a to b it creates/modifies the weight in G_s for (a,b).

Algorithm 1 SimpleGraph Gaze-Link

```
Require: Source code entities $SCE = \{e_1, e_2, \ldots, e_n\}$, set $S$ containing tuples of gaze data $(g_1, g_2, \ldots, g_n)$, where $g_i.entity \in SCE$ and $g_i.timestamp \in \mathbb{R}^+ \cup \{0\}$ is recorded by eye tracker, for each subject.
 1: for $s \in S$ do
 2:     { generate $G_s \in G$ from $g_i, g_{i-1} \in s$ }
 3:     if $g_i.entity \neq g_{i-1}.entity$ and $i > 1$ then
 4:         if $G_s[(g_i.entity, g_{i-1}.entity)] = 0$ then
 5:             $G_s[(g_i.entity, g_{i-1}.entity)] = (g_i.timestamp)^2$
 6:         else
 7:             $G_s[(g_i.entity, g_{i-1}.entity)] += g_i.timestamp$
 8:         end if
 9:     end if
10:     $NormalizeHighpass(0.95, L_s)$
11:     {generate $L_s \in L$ from $G_s \in G$}
12:     for $e \in SCE$ do
13:         $L_s[e] = \sum_{g \in \{g \in G_s | e \in g.key\}} g.value$
14:     end for
15:     $NormalizeHighpass(0.95, G_s)$
16:     {generate $C$ from $L_s \in L$}
17:     for $e \in SCE$ do
18:         $C[e] += L_s[e]$
19:     end for
20: end for
21: $NormalizeHighpass(0.5, C)$
22: return Composite linked entities $C = \{e_1 : \ell_1, e_2 : \ell_2, \ldots, e_m : \ell_m\}$ where $e_i \in SCE$ and $\ell_i \in \mathbb{R}^+ \cup \{0\}$
```

A constant offset equal to the minimum edge weight is subtracted from each edge in the graph and the maximum edge weight in the graph is used to normalize the weights to [0.0, 1.0] by dividing the weight of each edge by the maximum edge weight. At this point, the algorithm has determined source-to-source links from the developer's gaze data. A high-pass filter (*NormalizeHighpass*) with a cutoff of 0.95 is applied to the edges to eliminate extraneous links.

The end goal of the algorithm is to find task-to-source code entity links or use case-to- source code entity links. The SCE scores are therefore extracted from the remaining edges by summing the weights of all edges to which a given SCE is connected. Similar to the edge weights in the source-to-source link determination, the SCE scores are normalized to [0.0, 1.0] using the maximum score and are then run through a high-pass filter with a cutoff of 0.95.

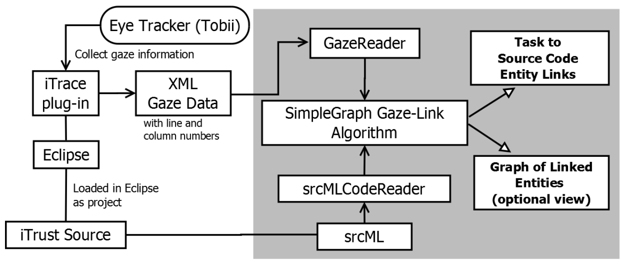
Eye Tracker (Tobii)
Collect gaze information
iTrace plug-in
XML Gaze Data
with line and column numbers
Eclipse
Loaded in Eclipse as project
iTrust Source
GazeReader
SimpleGraph Gaze-Link Algorithm
srcMLCodeReader
srcML
Task to Source Code Entity Links
Graph of Linked Entities (optional view)

Figure 1: An overview of the process used in generating links

SCE (linked to Task 1)	Score
`AddPatientFileAction.requiredFields`	1
`AddPatientFileAction.patients`	0.5
`CSVParser.CSVScanner`	0.5

Figure 2: Links generated for Task 1

The most important SCEs to the current task have now been determined for each developer. To get an answer reflecting all developers' activity, a composite of all developers' results for each task is generated. If an SCE appears in multiple developers' results, the scores are summed. Finally, the maximum and minimum scores are used to normalize the result to [0.0, 1.0] and a high-pass filter with a cutoff of 0.5 is applied. These SCEs and scores represent links from the SCEs to the current task and their related use cases with a strength proportional to the score. The algorithm is implemented in C# and run on the Mono framework. Our decision to use C# was primarily motivated by our intent to convert the experiment into TraceLab [3] components. We determined the cutoff values (0.95 and 0.5) via several test runs.

We now present an example of link generation. Consider the following task (task 1 in our study). The task was worded as follows.

Bug Description: The patient CSV uploader accessed from the "Upload Patient File" link from the Health Care Practitioner (HCP) sidebar does not behave as it should.
Expected: Uploading CSV with patient information requires the field firstName, lastName, and email
Actual: The firstName is not an acceptable field in CSV file
Related Use Cases: UC1, 1.3 Sub-flows S3, 1.6 Example CSV File

The correct place to fix the bug would be in *AddPatientFileAction.java* in the *requiredFields* variable. The *requiredFields* array was missing the firstname of the patient. When the algorithm was run on the developer eye gaze sessions, it generated the links (returned by the algorithm on line 22) presented in Figure 2.

4. THE PILOT STUDY

Our approach was empirically assessed in two stages. In the first stage, the developers' performed the bug fixing tasks. After the first stage was complete, we ran the SimpleGraph Gaze-Link algorithm on the gaze data collected from all four developers. The above section explains how the data was composited. This resulted in a set of links for each of the five tasks. In stage two, we asked another set of four developers to rank the set of links that the algorithm produces. We do this in two stages because the algorithm is currently run offline after all developers finish the tasks. The main research question we are seeking to answer is stated as follows: *Can eye gaze be used as input to retrieve traceability links?*

4.1 Eye-tracking and Data Collection

We used the Tobii X60 remote eye tracker that does not require the developer to wear any gear. *iTrace* connects directly with the eye tracker and gathers the raw data, i.e. (x,y) coordinates of where a developer is looking on the screen and the timestamp representing when the data was captured. We direct the reader to [8] for more information on how the plug-in interfaces with Eclipse. Gazes are only read when Eclipse's main window is in the foreground. For some subjects, *iTrace* picked up jumpy gaze data. To remedy this problem, new gazes are smoothed by averaging their positions with the positions of previous gazes.

Eye-tracking systems are particularly difficult when it comes to scrolling files larger than what fits on the screen. For example, Ali et al. [1] state that they use short segments of source code for their study to have better control over the eye-tracking system. Our environment does not have this limitation. *iTrace* supports scrolling both horizontally and vertically while maintaining the context of what the developer is actually looking at. *iTrace* correctly records the SCE that the developer is looking at as long as code folding is left off. To the best of our knowledge, *iTrace* is the first eye tracking environment that is able to achieve this accuracy with scrolling.

4.2 System, Tasks, and Subjects

The iTrust system (http://agile.csc.ncsu.edu/iTrust/) was used as the subject system in the study. A bug-localization task asks the developer to locate the code (class name and method/variable) that they would probably modify to fix the given bug. Each task contained a bug description, the expected behavior, the actual behavior, and all related use cases to the feature/requirement to where the bug was found. All five bugs were injected into a running version of iTrust (v. 15) that the subjects were able to use during the study. In other words, they were able to reproduce the bug given to them. Refer to Section 3 for an example of how a task was presented to the subjects. We collected difficulty and confidence levels for each task. A replication package with all study materials is available at http://www.csis.ysu.edu/~bsharif/itrace-pilot. In stage two, the task was to rank the list of SCEs (on a 5 point scale where 1 indicates lowest relevance, and 5 indicates highest relevance) that our algorithm produces for each task. The subjects were also asked to give a confidence value (low, medium, high) of their ranking. Note that they were not given the score produced by the algorithm for each SCE in order to keep their ranking unbiased. In the second stage, they were given the same material in stage one and in addition the correct place where the bug was fixed. They were also asked to report any other SCEs that were not listed. This would help uncover additional links they thought were relevant but that the algorithm did not find.

In stage one, a set of four expert developers (faculty and students) volunteered as subjects. These developers had an average of 3 years of experience using Java. All but one subject in stage one had experience in fixing bugs. All of the subjects in stage one are experienced programmers and have been actively programming for at least 10 years. In stage two, another set of four developers ranked the list of links based on their relevance to the task. Two of these developers also did stage one while the other two did not. These developers had at least 1 - 2 years of experience in using Java (two of whom have more than 5 years). All the subjects were familiar with using Eclipse and Java.

4.3 Study Instrumentation

A week before the study, we asked our participants to fill out a background questionnaire to collect information about their knowledge of Java, Eclipse, programming, and bug fixing skills. The participants in stage one were also sent a study instructions document, a tutorial to iTrust, and a sample bug localization task to familiarize them with the study format. During stage one, when subjects arrived at the lab, they were seated in front of a 24-inch LCD monitor and the moderator explained to them the entire study procedure. They also had a choice to go over the tutorials again. We then proceeded to calibrate the system to their eyes.

Each subject had a bug-injected copy of the iTrust application in Eclipse, related use cases, and a running version of the bug injected iTrust web application hosted on another lab machine. They were able to switch context back and forth between *iTrace* and the iTrust web application. *iTrace* only collects eye gaze data when the Eclipse window is in focus. We asked subjects not to resize the Eclipse window to maintain the same full screen setup for all subjects.

For each task, the subject was asked to click the Start and Stop tracking buttons available on the *iTrace* controller at the beginning and end of each task. This gave us an eye gaze session file saved for each task. They typed their answer (the class(es)/method (s)/attribute(s) where they might fix the bug) in a text file for each task within Eclipse. They were not required to change the code or fix the bug. Finally they completed an post-questionnaire.

For stage two, we printed booklets with the links generated by our SimpleGraph Gaze-Link algorithm for each task. The subjects were asked to rank these links on a 5-point scale. We gave them the location where the bug would have been fixed because we wanted them to rank the entities based on the correct answer and not their perception of the correct answer. All the bugs were one-line fixes. Besides the ranking of links we asked them to list SCEs they thought were relevant but not shown in the list. We provided them with an example on bug ranking to familiarize them with this procedure. They also completed a post questionnaire at the end.

5. STUDY RESULTS

Stage One: Of the four subjects participating in this stage, we received between one and four correct answers for any given task (see column $CRS1$ in Table 1). Here, correct answers mean that they were able to locate the location (i.e., class and method, or attribute) of the bug. Of incorrect responses, none of the methods or attributes given as responses contain a reference to the class containing the correct answer. It took 90 minutes on average to complete this stage.

Table 1: Experiment Results

T = True Links, EG = Number of SimpleGraph Eye-Gaze Links, $CRS1$ = Number of Correct Responses in Stage 1, $ALS2$ = Additional Developer Links in Stage 2, LAT = Links above Threshold, FP = False Positives, P = Precision, R = Recall, $Conf.$ = Confidence in task

	Stage 1		Algorithm Results			Stage 2					
Task	***CRS1***	***Conf.***	***T***	***EG***	***T ∩ EG***	***ALS2***	***LAT***	***Conf.***	***FP***	***P***	***R***
1	2	Med.	2	3	1	1	1	High	2	33%	50%
2	1	Low	4	5	0	3	2	Med.	5	0%	0%
3	3	High	1	3	1	2	1	High	2	33%	100%
4	2	High	2	4	1	0	1	Med.	3	25%	50%
5	4	Med.	2	7	1	1	3	Med.	6	14%	50%

Task 2, which received one correct answer, was rated as difficult by three of four subjects and average by the other subject. Task 5, which received all correct responses, was rated as average by three of four subjects and easy by one. In general, there was an inverse relationship between correctness and the subject's rating of difficulty (i.e., correct answers are rated less difficult).

SimpleGraph Gaze-Link Results : Given true links (T) and links that the algorithm found (EG), we calculated the algorithm's precision with $(T \cap EG) \div EG$ and its recall with $(T \cap EG) \div T$ for each task. EG represents the links found, given all developer's input to the algorithm. Section 3 explains this process. The results appear in Table 1. Task 2 is the only task which received both low precision and recall, which is attributed to the low amount of correct responses it received. Task 3 achieved a 100% recall with the rest achieving 50% recall.

Stage Two : Stage 2 assesses the usefulness of the automatically discovered links from eye gazes. The developers' ranking (in stage two) of the links the algorithm retrieved were averaged. Then, we set a threshold of 2.5 (individual ranks are integers $\in$ [1, 5]) and filtered out links below this value. The count of links remaining for each task appears in Table 1 as LAT. An LAT value above 0 means the algorithm is finding at least one relatively useful link.

6. OBSERVATIONS AND DISCUSSION

In all but task 2, the correct class appears in the list of candidate links. For all but task 5, the correct method or attribute appears. Therefore, if a correct link is not found at the method/attribute granularity, there is a high likelihood that it is found at the class granularity. This observation shows that it is feasible to use eye gaze to retrieve traceability links. The confidence (see Table 1) was always higher in stage two compared to stage one for each task. From reviewing the post-questionaire given in stage one, subjects said they had enough time to complete the study and rated it as difficult overall. When asked if they have experience fixing bugs using the information from a bug report, one subject said yes, another said no, and the others said sometimes. Subjects from stage two stated that they had sufficient time to complete the study and that if they were asked to complete the bug task, they would find the entities suggested by the SimpleGraph Gaze-Link algorithm useful. Overall, the study's difficulty level was average in the second stage.

It is possible that some bias exists with respect to developers' eye gazes. We plan on conducting a thorough sensitivity analysis of the bias as part of our on-going investigation. Since this study was a pilot, the tasks selected were restricted to one-line bug fixes and were created by us. We will include multiple-line/class fixes for real bug reports as part of our next set of experiments. One developer solved Task 2 correctly in Stage 1 (see column CRS1 in Table 1); however, the SCEs that were retained by the SimpleGraph Eye-Gaze algorithm did not include the correct answer resulting in a zero recall and precision. This glitch suggests tuning the parameters in our algorithm to respond better to more complex tasks. We also plan on directly comparing our algorithm with existing techniques (e.g., LSI) to assess their mutual benefits. The captured traces to bug tasks will eventually be used in a machine learning approach to inform future traces on similar/related bugs across stakeholders.

Mylyn bridges issue tracking systems and Eclipse. With this support, Eclipse can monitor all the developer's activities for a specific task (e.g., bug fix). For example, it can record the files that are opened and changed (termed as Mylyn events) to fix a specific bug. Mylyn recorded events may offer an alternative to eye tracking measures, albeit, may not be as detailed and concise as offered by eye gazes. Mylyn gives change events but not the added rationale that led to those events, which is available from an eye gaze. For example, a developer could have looked at three elements in one artifact before making a change. Mylyn will record the change and not the eye movements that preceded it. These eye movements could indicate potential links between items viewed/changed. Indeed, it is feasible to use our approach in addition to Mylyn, as both collect different types of data.

7. RELATED WORK

We first introduced the notion of using eye tracking for software traceability tasks in [7]. In 2013, we presented our initial eye tracking framework, *iTrace* [8], our second paper where we set the stage to conduct traceability experiments in the future. Both position papers above did not have a formal algorithm as to how to go about retrieving links nor did they conduct a study to determine the feasibility of the approach, which this paper addresses.

The closest work to using eye tracking in the software traceability field is the work by Ali et al. [1]. Their work comes after our initial presentation of the use of eye tracking in software traceability in 2011. They used eye tracking on small snippets of code to determine what a developer is looking at most. They found that developers tend to look more at method names than class names. They also ask developers to rank source code entities (class, method, comments, variable names). Based on this finding, they propose two weighting schemes for LSI (IR-based method) to retrieve traceability links from the most frequently used data sets: iTrust (v. 10) and Pooka (v. 2). Our work presented here relies solely on eye tracking data of developer sessions to derive links from tasks to code. Ali et al. [1] do not use eye tracking data to directly derive links nor do they use eye tracking on the source code of subject systems. Also, their ranking is on which SCE is preferred, our ranking is on validating traceability links generated from our algorithm. This is in contrast to our work which is developer-centric rather than artifact-centric. We are not aware of any other work that is using eye tracking in traceability at the time of this writing.

8. CONCLUSIONS AND FUTURE WORK

We present an algorithm that automatically determines links based on developers' eye gazes. We validate these links by asking another set of developers to rank them based on their relevance to the bug tasks. We also compare the links with a set of true links. The results are particularly promising in terms of recall. Precision needs improvement; however, it could be an attribute of the small scale of the study. *iTrace* provides a novel platform that would directly support two key software traceability tasks: traceability link generation/recovery, and traceability link maintenance and evolution. Additionally, it provides a new methodology for researchers to conduct empirical studies in the software traceability domain. *iTrace* is still under development and will be released as open-source once a stable version is reached. In the future, we plan on refining the algorithm and conducting larger scale studies. In this work, we only focused on source code files, but we also have the data for other artifacts such as links to jsp files as well as use cases, which we plan to analyze in the future. We are also in the process of incorporating the algorithm into a Tracelab [3] component.

9. REFERENCES

[1] N. Ali, Z. Sharafi, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study on requirements traceability using eye-tracking. In *ICSM*, pages 191 – 200, 2012.

[2] H. Asuncion, F. Francois, and R. N. Taylor. An end-to-end industrial software traceability tool. In *6th ESEC/FSE*, pages 115–124, 2007.

[3] J. Cleland-Huang, A. Czauderna, A. Dekhtyar, O. Gotel, J. H. Hayes, E. Keenan, G. Leach, J. I. Maletic, D. Poshyvanyk, Y. Shin, A. Zisman, G. Antoniol, B. Berenbach, A. Egyed, and P. Maeder. Grand challenges, benchmarks, and tracelab: developing infrastructure for the software traceability community. In *6th TEFSE*, pages 17–23, 2011.

[4] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artefact management systems using information retrieval methods. *ACM (TOSEM)*, 16(4,):article no. 13, 2007.

[5] P. Mader, O. Gotel, and I. Philippow. Motivation matters in the traceability trenches. In *RE 2009*, pages 143–148.

[6] B. Ramesh and M. Jarke. Towards reference models for requirements traceability. *IEEE TSE*, 27(1):58–93, 2001.

[7] B. Sharif and H. Kagdi. On the use of eye tracking in software traceability. In *6th TEFSE*, pages 67–70, 2011.

[8] B. Walters, M. Falcone, A. Shibble, and B. Sharif. Towards an eye-tracking enabled ide for software traceability tasks. In *7th TEFSE*, pages 51–54, 2013.