Interlacing Gaze and Actions to Explain the Debugging Process

Kshitij Sharma, Norwegian University of Science and Technology, kshitij.sharma@ntnu.no
Katerina Mangaroska, Norwegian University of Science and Technology, katerina.mangaroska@ntnu.no
Michail Giannakos, Norwegian University of Science and Technology, michailg@ntnu.no
Pierre Dillenbourg, Ecole Polytechnique Fédérale de Lausanne, pierre.dillenbourg@epfl.ch

Abstract: Debugging is an indispensable skill of successful programmers. As such, teacher's should not overlook it when teaching programming. The main aim of the study is to use gaze data combined with measures at different temporal granularities to show how these measures are related to the outcomes (in particular debugging success as a learning by doing outcome) students have at the end of the debugging task. The results delineate that combining gaze data with actions (reading, writing, scrolling) and unit tasks (main method and JUnit test) gives new insights to further understand the cognitive actions in debagging a program. Moreover, this study also focuses on discovering debugging patterns of successful students in order to improve the design of learning activities to teach students how to debug. Finally, with the analysis, the authors have shown an automatic way of detecting successful action-gaze patterns.

Keywords: Computer science education, Eye-tracking, Time-scales, Program debugging, Programming.

Introduction

One of the most essential 21st century skills is programming. We have witnessed the influence of programming in disciplines like engineering, biology, chemistry, etc. Therefore, many engineering programs require students to take introductory programming classes. However, even today there is still no solid retention in programming courses. In addition, (McCracken et al., 2001) assessed the skills and competences of first year programming students concluding that many freshman cannot write correct code. One of the reasons behind this issue is the difficulties and challenges students face while learning to program. An example of this is debugging. Besides being vital, learning to debug is difficult and requires application of many skills simultaneously that majority of freshman do not have at the beginning (Perkins & Martin, 1986). However, debugging is an indispensable skill of successful programmers and cannot be neglect it when teaching programming. Moreover, there are no best practices or pedagogies available for programming educators to teach "*how to debug*" (Fitzgerald et al., 2010). Another reason is lack of computational thinking skills. Computational thinking is a fundamental skill for everyone, not just for programmers (Wing, 2006). It is way humans solve problems, by thinking at multiple levels of abstraction. It also means applying heuristic reasoning in presence of uncertainties when solving complex problems. As such, computational thinking should be part of the skill set of future programmers. Consequently, if programming educators want to improve the curriculum and the teaching, they need to understand how students solve problems, what resources they use, what strategies they follow, and what motivates them to continue to learn programming. Eye-tracking can help them to get closer to the students' way of learning programming and observe actions that cannot be captured by observations or through system log data.

Eye-tracking allows researchers to get attentional data for users while they perform tasks at a time scale that has more information content than other measures like event logs, dialogues, or gestures (Sharma, Jermann, Nüssli, & Dillenbourg, 2012). (Newell, 1994) proposed time scales and human action bands to describe behavior at various levels. For example, the duration of gaze fixations is usually 100 milliseconds, placing it at the lower cognitive band. Lower cognitive bands are the primitive actions (Newell's Deliberate Acts) that Newell believed if combined the right way at this level can lead to understanding the development of higher level constructs, such as expertise. This has been supported in studies like Gluck's study of instructional leverage with an algebra tutor (Gluck, 1999) or Ackerman's study of processing parallel threads in air traffic controller task (Ackerman, 1994). At the higher end, cognitive bands require complex actions (e.g. reading or gestures). Since there is a time gap between significant educational outcomes (i.e. tens of hours to achieve) and effects measured in tens of milliseconds, (Anderson, 2002) identifies cognitive modeling as bridging across the behavioral bands by taking the lower level bands into account. The main aim of this study is to use gaze data to define measures at different temporal granularities, and show how these measures are related to the outcomes (in particular debugging success as a learning by doing outcome) students have at the end of the debugging task. In addition, we will use screen recording from the eye-trackers to disambiguate the actions into reading, writing and scrolling episode, and find patterns that successful students follow. These patterns could later indicate opportunities for instructional leverage. In this contribution the study was driven by the following research question: **"How the interplay between the gaze patterns and actions explains the debugging success?"**

The rest of the paper is organized as follows: the second section is the related work for the present study. The third section describes the research methodology. The fourth section describes the various variables used in the experiment. The fifth section presents the analysis results. Finally, the sixth section discusses the results and concludes the paper.

Related work

Eye-tracking and education

Studying successful and unsuccessful programmers provide important insights for improving teaching because it allows to understand how programmers learn and what misconceptions they might have. (Vessey, 1985) found out that expert debuggers were following a breadth first approach, trying to understand a program and build mental representations, while novices were following depth first approach, focusing on finding errors rather than understanding the program. Consequently, if programming educators have such insights, it might help them to design correct models "*how to teach novices*". Adapting to the technological advancements, gaze data has already proven its value in understanding how students learn to program and debug (Sharma, Jermann, Nüssli, & Dillenbourg, 2013). Eye-tracking helps researchers to gain insights into user behavior (e.g. how the user process information or interacts with visual information) that cannot be captured verbally, via other more ordinal user-data (e.g. click-stream) or with observations (Cooke, 2005). Using eye-tracking to investigate differences between experts and novices, but also between "good" and "poor" novices, could give new insights that could be used to develop instructions for a specific group of learners. In addition, researchers could also gain further understanding what is happening in the stages of understanding the program, testing the program, or locating the errors in the program.

Eye-tracking and debugging

Most results show that eye-tracking allows researchers to get users' attention, while they perform tasks to explain various constructs like contextual expertise, task-based performance, and task complexity (Harbluk, Noy, Trbovich, & Eizenman, 2007; Kaller, Rahm, Bolkenius, & Unterrainer, 2009; Reingold, Charness, Pomplun, & Stampe, 2001). Furthermore, ''debugging is a skill that does not immediately follow from the ability to write code. Rather…it must be taught'' (p. 208) (Kessler & Anderson, 1986). In addition, a comprehensive review of debugging in education has been performed; however, little has been done after it to gain new insights (McCauley et al., 2008).

Previous studies (Bednarik, 2012; Bednarik & Tukiainen, 2004, 2008) showed a clear relation between gaze patterns and task-based performance in debugging, underlying the importance of using eye-tracking to understand students' patterns when running into difficulties while working on problem-solving tasks (e.g. debugging code). Thus, in order to avoid programming errors that result from misconceptions, programming educators should not ignore incorrect mental models students might have, but must understand their constructs that provide insights how to design curriculums to teach programming/ debugging techniques (McCauley et al., 2008). For example, in (Sharif, Falcone, & Maletic, 2012) authors compared the first scan time (the time takes by the participants to read the code for the first time) against the different levels of debugging success. The results showed that successful debuggers had a significantly lower first scan time than the less successful ones. In terms of gaze behavior, this study showed that successful debuggers had a more vertical gaze than those who perform less successfully during the debugging task. Moreover, (Stein & Brennan, 2004) in a study considering a collaborative debugging task, showed that a gaze-contingent condition (to find a bug after viewing gaze trace as a visual cue) induced a better way for finding bugs in a given code. Like-wise, (McDowell, Werner, Bullock, & Fernald, 2006) came to the same conclusion years before, unveiling that students working in pairs are more successful, more confident, and more likely to continue studying programming. One common drawback of most of the previous studies (to the best of our knowledge) concerning debugging, is that the task was limited to "*find the bug*" and not actually to "*fix the bug*". This limits these studies to a mere comprehension task. In the present study we asked the students to *"find and fix the bug"*. Thus, our study captures more in-depth data (both in terms of actions and gaze) in a debugging task.

Methodology

The debugging activity

The authors designed and implemented a debugging activity in conjunction with the partners from the École Polytechnique Fédérale de Lausanne University. The main task assigned to the participants was debugging a code that contained consistency issues implemented as unit tests. The consistencies that were absent from the original version of the code provided to the participants are the following: 1) Gender consistency: the mother should be a female and the father should be a male; 2) Child-parent consistency: if Jens is the child of Merit, Merit should be the mother of Jens; and vice-versa; 3) The removal of a child- parent relationship from either a parent or a child should also apply to the whole family; 4) Adoption consistency: the child-parent (addition and removal) and the gender consistencies should be maintained in the case of an adoption.

Participants

During the spring 2017, an experiment was conducted at a contrived computer lab setting at École Polytechnique Fédérale de Lausanne University with 40 computer science majors (12 females and 28 males) in their third semester. The mean age of the participants was 19.5 years (Std. Dev. = 1.65 years). In the previous semester, all of the participants had taken a Java course, where they were predominantly using Eclipse as Integrated Development Environment (IDE). Moreover, they were also familiar with the built-in debugging tool provided by Eclipse.

Procedure

Upon arrival in the laboratory, the participants signed an informed consent form. After this and prior to the debugging task, each participant had to pass an automatic eye-tracking calibration routine to accommodate the eye tracker's parameters to each participant's eyes to ensure accuracy in tracking the gaze. Their gaze during the debugging task was recorded using an SMI RED 250 eye-tracker at 250 Hz. Next, the participants were asked to perform a pre-task, which required removing 90 errors from a skeleton code within ten minutes. After this task, the participants were given 40 minutes to solve five debugging tasks presented as a part of the main method of the main class of a 100 lines of Java code. The code for the main debugging task contained no syntactic errors, and the participants were notified about this fact. For their participation in the experiment, the participants were rewarded with an equivalent of USD 30.

Variables

Debugging success

For the debugging task, there were ten unit tests prepared by the instructor. These unit tests were about the consistency of the parent-child relationship (see Subsection "The debugging procedure"). To limit the debugging to one of the panels of the Eclipse IDE, the researchers introduced few bugs in otherwise complete code that would make the code fail all ten unit tests. In order to pass all of the unit tests, the students were required to solve the debugging exercises in a particular order. For example, the gender consistency needed to be solved before the child-parent consistency. Participants were given 40 minutes to complete the task. At the end of the 40 minutes, they were told to stop, and the number of unit tests passed at that point of time was taken to be the measure of the "debugging success". Hereafter we will refer to "debugging success" as "success".

Unit tasks: JUnit tests and main method

In order to complete the debugging task, the participants were testing their solutions multiple times. They could do this in two different ways: 1) by running the set of JUnit tests, or 2) by executing the main method of the program. We divided the whole debugging session into two types of episodes based on the unit tasks performed by the participants namely, **JUnit** and **main**.

Actions: Reading, writing, and scrolling episodes

Participants' screen was recorded during the debugging task (at 10 frames per second). We computed framewise image difference, in order to detect reading, writing, or scrolling episodes. First, we assigned, to each frame, an action flag denoting whether it was a reading, writing, or scrolling frame. For doing so, we used two thresholds for the number of pixels changed across two frames. If there were no pixels changed, we assigned the frame a **reading** flag; if the change was corresponding to a range of 0.4 -- 0.8 characters changed over two frames, we assigned the frame a **writing** flag; for a change corresponding to more than 10 characters per frame we assigned the frame a **scrolling** flag. For obtaining 0.4 pixels to 0.8 pixels, for the writing labels, we used the following logic. The average person's typing speed is 40 words per minute, the average length of each word in JAVA could be taken as 6 characters. This translates to 240 characters per minute, or 4 characters per second, or 0.4 characters per frame. We doubled this limit to accommodate for the fast typing participants as well.

After assigning reading, writing or scrolling flags, we computed the proportionality vector from the counts of action flag from 5 seconds (50 frames). Finally, we used a winner-take-all strategy to assigned each 5 seconds episode a reading, writing, or scrolling label.

Gaze: Entropy-stability episodes

To define the entropy and stability episodes, we first overlaid a 50-pixels-by-50-pixels grid (hereafter referred to as grid) on the screen and we divided the whole debugging session into 10 second time windows (hereafter referred to as window). We then computed the proportion of time spent on each block in the grid. Using a two-dimensional proportionality vector, we computed the entropy and stability measures as follows:

Entropy: the entropy is calculated as the Shannon entropy of the proportionality vector. This measure tells us about the *focus size* of the participant. A value of 0 will show that the participant was looking at only one block on the grid. In other words, entropy measures the level of uncertainty of a random variable, which is, in our case, the objects looked at by the subjects. Theoretically, the highest possible entropy value is *log (number of blocks in the grid).* In our case it is *2.76.* This value will depict a uniform distribution of time over the grid. Thus, a high value of entropy would mean that the participant was looking at a wider range of objects on the screen; in other words, the participant had a higher size of focus (i.e. *not focused gaze*). Low entropy indicates that they mostly looked at few objects (i.e. *focused gaze*). Finally, it is important to point out that the "focus size", in theory, is not at all related to "attention level". It merely captures the number of objects the participant is looking at in a fixed time window.

Stability: the stability is computed over two successive windows. This measure tells us about the *similarity between the objects* looked by the subject and the duration of the gaze. We compute the cosine similarity between the two windows. This value is bounded between 0 and 1 (both included). A stability value of 1 will depict that the participant was looking at the same set of objects during two consecutive time windows, while a stability value of 0 will show that the participant was looking at completely different set of objects during two consecutive time windows.

Once, we had computed the entropy and stability for the participant, we applied a median cut to define focused and unfocused windows (based on entropy) and stable and unstable windows (based on stability). Finally we used a run-length encoding to define four types of gaze episodes: **focused-stable, focused-unstable, unfocused-stable, and unfocused-unstable**.

Gaze:Transitions

We divided the whole Eclipse IDE into 8 functional Areas Of Interest (AOI) as follows.

1. **Variable View (VV)**: During debugging, allows changing the value of a variable to test how your program handles a particular value or to speed through a loop.
2. **Debug View (DV)**: Manages the debugging or running of a program in the workbench.
3. **Project Explorer (PE)**: Provides a hierarchical view of the artifacts in the workbench.
4. **JUnit (JU)**: Lists the unit tests to be passed by the main Java class.
5. **Exercise View (EV)**: Allows seeing the coding, saving, testing, and progress.
6. **Problem (Pr)**: Shows the errors and warnings raised by the Java Compiler.
7. **Console (Cn)**: Shows the output of the code.
8. **Code (Co):** Panel where the code is written.

We then computed the gaze shifts from one AOI to another and grouped them following pre-defined categories: Locate problem, Locate variable, Hypothesis verification, Fix problem, Read code.

Results

Performance, unit-tasks, entropy-stability episodes

1) While in a main method running episode the success is positively correlated to the time that the participants spent as focused but unstable. 2) While in a junit test running episode the success is positively correlated to the time that the participants spent as focused and stable. 3) While in a junit test running episode the success is negatively correlated to the time that the participants spent as unfocused and unstable. Table 1 shows the

different linear models, and the Figure 1 shows the significant correlations as mentioned at the starting of this paragraph.

Table 1: Linear models for the success using different unit-tasks and entropy-stability episodes. Var. = variable; est.= estimate; JU = Junit; Fo = focused; Uf = unfocused; S = stable; Us = unstable; * = p-value (0.05); ** = p-value (0.01); *** = p-value (0.001)

Var.	est.	error	t-val.	Var.	est.	error	t-val.	Var.	est.	error	t-val.
Int.	3.8	0.9	4.2***	**Int.**	2.9	0.9	3.1***	**Int.**	2.6	1.0	2.5**
JU	0.7	1.2	05	**JU**	-2.3	1.2	-1.9	**JU**	2.7	1.4	1.9
Main	-1.5	1.1	-1.3	**Main**	0.5	1.1	0.5	**Main**	0.7	1.4	0.5
FoUs	6.3	3.2	1.0	**FS**	13.0	6.7	1.9	**UfS**	6.5	7.8	1.1
JU:FoUs	-6.1	8.6	-0.7	**JU:FS**	16.3	8.1	2.1*	**JU:UfS**	-23.5	10.4	-2.2*
Main:FoUs	16.5	8.0	2.1*	**Main:FS**	-0.9	8.9	-0.1	**Main:UfS**	-6.9	10.6	-0.6

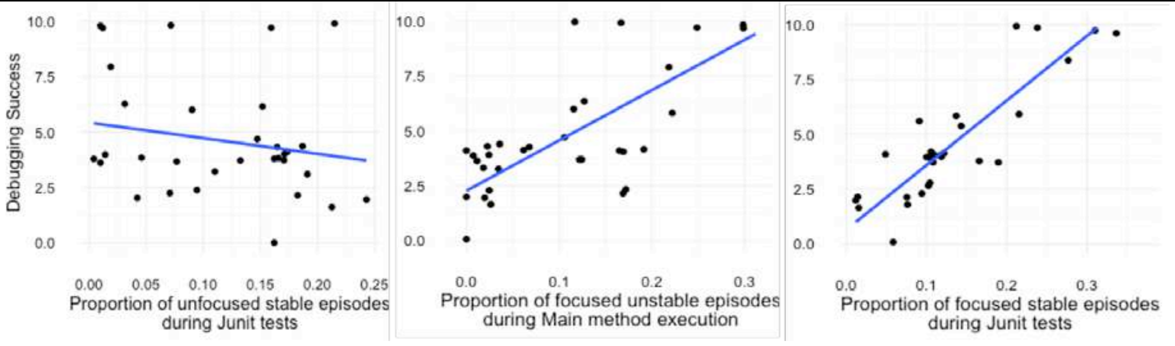

Figure 1. Debugging success for the different gaze episodes and unit-tasks, the line shows the linear model.

Performance, unit-tasks, transitions

1)While in a junit test running episode the success is positively correlated to the proportion of locate problem transitions. 2) While in a junit test and main method running episode the success is positively correlated to the proportion of locate variable transitions. 3) While running the main method the main sub-task was to produce a desired output this could lead to "hypothesis generation - verification" loop and they go back and forth between code and console many times. 4) While in a junit test running episode the success is correlated to the proportion of fix problem transitions. Table 2 shows the different linear models, and the Figure 2 shows the significant correlations as mentioned at the starting of this paragraph.

Table 2: Linear models for the success using different actions and transitions. Var. = variable; est.= estimate; err. = error;JU = JUnit test;   LP = locate problem; LV = locate variable; HV = hypothesis verification; FP = fix probelm; * = p-value (0.05); ** = p-value (0.01).

Var.	Est.	Err.	t-val	Var.	Est.	Err.	t-val
Int	5.3	0.9	5.4**	**Int**	3.6	0.91	3.9**
Main	-1.1	1.3	-.08	**Main**	-1.6	1.1	-1.4
JU	-2.3	1.3	-1.1	**JU**	-2.2	1.1	-1.9
LP	-7.2	8.6	-0.8	**LV**	8.9	7.6	1.1
Main:LP	12.3	12.5	0.9	**Main:LV**	18.7	9.2	2.1*
JU:LP	22.2	11.0	2.1*	**JU:LV**	21.1	9.5	2.2*
Var.	**Est.**	**Err.**	**t-val**	**Var.**	**Est.**	**Err.**	**t-val**
Int	4.5	0.8	5.4**	**Int**	4.7	1.1	4.3**
Main	-2.7	1,1	-1.0	**Main**	-1.0	1.3	-0.7
JU	-0.3	1.3	-0.2	**JU**	-3.1	1.4	-1.8
HV	0.8	7.0	0.1	**FP**	-1.0	8.4	-0.1
Main:HV	20.7	8.4	2.4**	**Main:FP**	11.9	11.3	1.0
JU:HV	3.6	11.8	0.3	**JU:FP**	36.4	12.8	2.8**

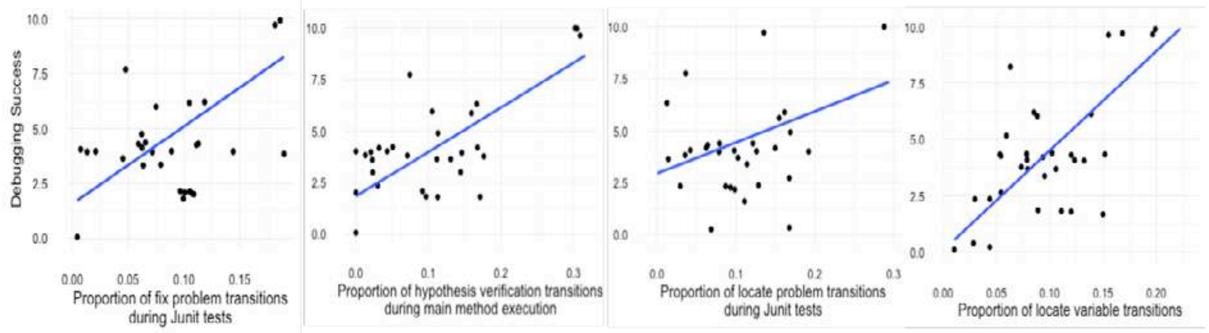

Figure 2. Debugging success for the different gaze transitions and unit-tasks, the line shows the linear model.

Performance, actions, entropy-stability episodes

1) While in a writing episode, the successful participants are focused stable. 2) While in a scrolling episode, the successful participants are unfocused unstable. 3) While in a reading episode, the successful participants are focused unstable. Table 3 shows the different linear models, and the Figure 3 shows the significant correlations as mentioned at the starting of this paragraph.

Table 3: Linear models for the success using different actions and entropy-stability episodes. Var. = variable; est.= estimate; Fo = focused; Uf = unfocused; S = stable; Us = unstable; * = p-value (0.05); ** = p-value (0.01)

Var.	**est.**	**error**	**t-val.**	**Var.**	**est.**	**error**	**t-val.**	**Var.**	**est.**	**error**	**t-val.**
Int	3.5	8.1	4.4**	**Int**	5.4	0.9	5.8**	**Int**	4.3	1.1	3.8**
read	1.4	1.4	1.0	**read**	-0.7	1.4	-0.4	**read**	-2.8	1.4	-1.8
scroll	1.3	1.1	0.1	**scroll**	-2.3	1.2	-1.8	**scroll**	1.9	1.5	0.1
write	-1.9	1.1	-1.5	**write**	-0.3	1.3	-.02	**write**	0.2	1.5	0.1
FS	9.8	6.3	1.5	**UfUs**	-6.8	6.1	-1.1	**UfS**	2.1	7.5	0.2
read:FS	-1.9	9.5	-1.3	**read:UfUs**	5.6	9.7	0.5	**read:UfS**	21.6	9.8	2.2*
scroll:FS	-4.3	8.9	0.1	**scroll:UfUs**	18.3	7.9	2.3*	**scroll:UfS**	-0.1	10.6	0.1
write:FS	15.4	7.4	2.1*	**write:UfUs**	2.6	9.0	0.2	**write:UfS**	-1.9	10.2	-0.9

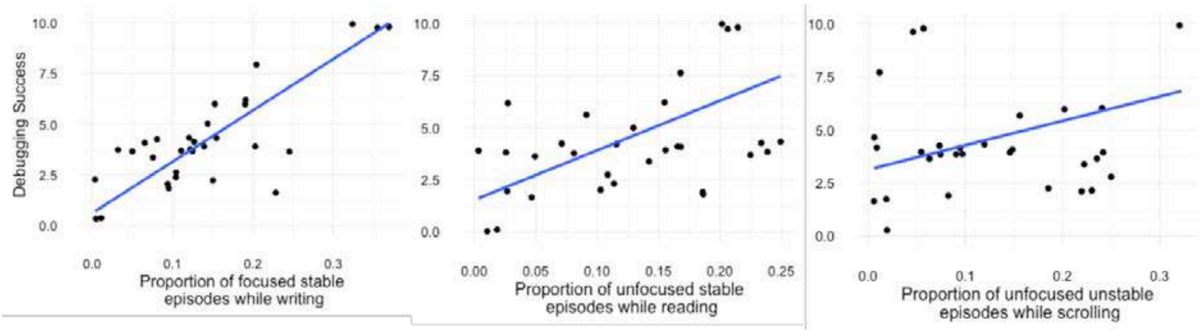

Figure 3. Debugging success for the different gaze episodes and actions, the line shows the linear model.

Performance, actions, transitions

During reading, writing, and scrolling episodes success is positively correlated to locate problem, hypothesis verification, and locate variable transitions, respectively. Table 4 shows the respective linear models.

Table 4: Linear models for the success using different actions and transitions. Var. = variable; est.= estimate; LP = locate problem; LV = locate variable; HV = hypothesis verification; ** = p-value (0.01)

Var.	**est.**	**error**	**t-val.**	**Var.**	**est.**	**error**	**t-val.**	**Var.**	**est.**	**error**	**t-val.**
Int	5.3	1.1	5.0**	**Int**	5.4	0.8	6.1**	**Int**	5.4	0.9	6.0**
read	-3.8	1.3	-1.8	**read**	-0.9	1.2	-0.7	**read**	03	1.5	0.2
scroll	0.009	1.4	0.1	**scroll**	-4.2	1.2	-1.4	**scroll**	0.009	1.2	0.1

write	-2.1	1.3	-1.5	**write**	-1.2	1.3	-0.9	**write**	-4.4	1.2	-1.6
LP	-7.8	10.8	-0.7	**LV**	-8.9	7.9	-1.1	**HV**	-8.5	7.8	-1.1
read:LP	31.1	11.3	2.7**	**read:LV**	10.2	10.6	0.9	**read:HV**	-1.2	12.1	-0.1
scroll:LP	-0.004	14.3	0.1	**scroll:LV**	33.0	9.2	3.5**	**scroll:HV**	0.005	11.8	0.1
write:LP	2.2	13.0	1.7	**write:LV**	12.7	11.7	1.1	**write:HV**	33.5	9.4	3.7**

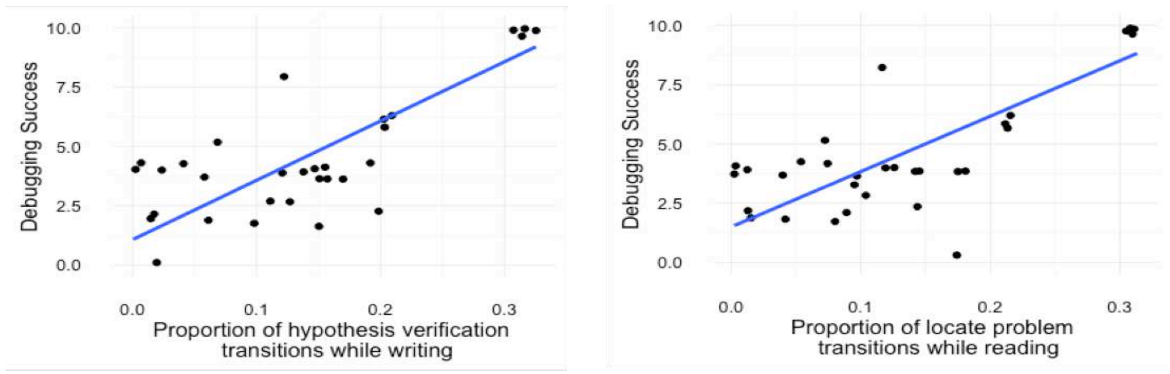

Figure 4. Debugging success for the different gaze transitions and actions, the line shows the linear model.

Discussion and Conclusions

In this contribution, we study the relation between the gaze patterns and the actions, each at two different levels, to explain the difference between successful and unsuccessful debuggers. We defined two gaze variables: focus-stability episodes and transitions; and two action variables: unit-tasks and reading/writing/scrolling episodes. Our results show that these variables, when combined together, can enable us to understand debugging approaches, that extend beyond the state-of-the-art "finding bugs" approaches. This is one way that can help programming educators to capitalize on the debugging tools to develop novel instruction to teach students how to debug (McCauley et al., 2008).

Combining the unit tasks (main method execution and JUnit test) with the gaze variables, we observed that the students follow two different trajectories during the two different unit tasks. While in the main method execution episode the successful students locate the problem in the code and perform a hypothesis generation-verification cycle. This results in a focused but unstable gaze for two reasons. First, they have to look at different small parts of the program to locate the problem; and second, to perform the hypothesis generation-verification they have to repeatedly switch between the code and the output resulting in looking at two different (unstable gaze) but small sets of objects looked at (focused gaze). On the other hand, the JUnit test descriptions are designed to be precise. The successful students use this description to locate the problem causing variable and to fix it. To do so, the students often concentrate on a small part of program (focused gaze) for a longer duration (stable gaze).

Combining the actions (reading, writing, scrolling) with the gaze variables, the data revealed three different strategies from the successful students. First, while reading, the successful students try to locate the problem as a result, and as explained before, they have a focused-unstable gaze. Second, while writing, the successful students fix the problem using a hypothesis generation-verification cycle, and in turn they exhibit a focused-stable gaze, similar to the previous paragraph. Finally, when scrolling the students perform a typical search for the problematic variable and hence have a unfocused-unstable gaze. One can argue that this relation between gaze and action patterns is not surprising. However, with our analysis, we have shown an automatic way of detecting the successful action-gaze pattern. In future, this can be leveraged to design and develop real-time automated tools to help the students while debugging a program.

The findings reported in the second paragraph are not obvious and were reached by analyzing gaze data. However, many researchers might argue that the findings reported in the third paragraph were obvious, but we would argue that those findings are intuitive but not obvious. As it can be seen through the study design, we let the students to write the code, which makes eye-tracking analysis not-straightforward, since the stimulus changes with time and is different for each student.

In a nutshell, we showed that combining gaze data with actions (reading, writing, scrolling) and unit tasks (main method execution and JUnit test) help us to further understand the cognition that underlies debugging a program and discover debugging patterns of successful students. Moreover, allowing students to *"find and fix the bug"* helped us to capture more in-depth data in terms of actions and gaze during the debugging task. Adapting to the technological advancements and using eye-tracking to study successful and unsuccessful programmers provide important insights for improving teaching and learning how to teach debugging as a skill successful programmers need to have it. The results are interesting enough to pursue further research to observe how and when novices build basic knowledge about variables and expressions, various mental models, as well as what misconceptions cause what type of programming errors.

References

Ackerman, P. (1994). Kanfer-Ackerman air traffic controller task© CD-ROM database, data collection program, and playback program. *Office of Naval Research, Cognitive Science Program*.

Anderson, J. R. (2002). Spanning seven orders of magnitude: A challenge for cognitive modeling. *Cognitive Science, 26*(1), 85-112.

Bednarik, R. (2012). Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *International Journal of Human-Computer Studies, 70*(2), 143-155.

Bednarik, R., & Tukiainen, M. (2004). *Visual attention tracking during program debugging.* Paper presented at the Proceedings of the third Nordic conference on Human-computer interaction.

Bednarik, R., & Tukiainen, M. (2008). *Temporal eye-tracking data: evolution of debugging strategies with multiple representations.* Paper presented at the Proceedings of the 2008 symposium on Eye tracking research & applications.

Cooke, L. (2005). Eye tracking: How it works and how it relates to usability. *Technical Communication, 52*(4), 456-463.

Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., & Zander, C. (2010). Debugging from the student perspective. *IEEE Transactions on Education, 53*(3), 390-396.

Gluck, K. A. (1999). Eye movements and algebra tutoring.

Harbluk, J. L., Noy, Y. I., Trbovich, P. L., & Eizenman, M. (2007). An on-road assessment of cognitive distraction: Impacts on drivers' visual behavior and braking performance. *Accident Analysis & Prevention, 39*(2), 372-379.

Kaller, C. P., Rahm, B., Bolkenius, K., & Unterrainer, J. M. (2009). Eye movements and visuospatial problem solving: Identifying separable phases of complex cognition. *Psychophysiology, 46*(4), 818-830.

Kessler, C. M., & Anderson, J. R. (1986). *A model of novice debugging in LISP.* Paper presented at the Proceedings of the First Workshop on Empirical Studies of Programmers.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education, 18*(2), 67-92.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., . . . Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33*(4), 125-180.

McDowell, C., Werner, L., Bullock, H. E., & Fernald, J. (2006). Pair programming improves student retention, confidence, and program quality. *Communications of the ACM, 49*(8), 90-95.

Newell, A. (1994). *Unified theories of cognition*: Harvard University Press.

Perkins, D., & Martin, F. (1986). *Fragile knowledge and neglected strategies in novice programmers.* Paper presented at the first workshop on empirical studies of programmers on Empirical studies of programmers.

Reingold, E. M., Charness, N., Pomplun, M., & Stampe, D. M. (2001). Visual span in expert chess players: Evidence from eye movements. *Psychological Science, 12*(1), 48-55.

Sharif, B., Falcone, M., & Maletic, J. I. (2012). *An eye-tracking study on the role of scan time in finding source code defects.* Paper presented at the Proceedings of the Symposium on Eye Tracking Research and Applications.

Sharma, K., Jermann, P., Nüssli, M.-A., & Dillenbourg, P. (2012). *Gaze Evidence for different activities in program understanding.* Paper presented at the 24th Annual conference of Psychology of Programming Interest Group.

Sharma, K., Jermann, P., Nüssli, M.-A., & Dillenbourg, P. (2013). *Understanding collaborative program comprehension: Interlacing gaze and dialogues.* Paper presented at the Computer Supported Collaborative Learning (CSCL 2013).

Stein, R., & Brennan, S. E. (2004). *Another person's eye gaze as a cue in solving programming problems.* Paper presented at the Proceedings of the 6th international conference on Multimodal interfaces.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies, 23*(5), 459-494.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35.