Middle School Students Using Alice: What Can We Learn from Logging Data?

Linda Werner
University of California
Santa Cruz, CA
linda@cs.ucsc.edu

Charlie McDowell
University of California
Santa Cruz, CA
charlie@cs.ucsc.edu

Jill Denner
Education, Training, Research Assoc.
Scotts Valley, CA
jilld@etr.org

ABSTRACT
There is growing interest in how we can use computer logging data to improve computational tools and pedagogies to engage children in complex thinking and self-expression, but our techniques lag far behind our theories. Only recently have learning scientists begun to measure, collect, analyze, and report how data informs the science of children's learning. In this paper, we describe our initial efforts towards developing tools to mine computer logging data for information on how to enhance learning opportunities. The data were collected as part of an NSF-funded project, and include logs from 320 middle school students using Alice to program computer games in semester-long courses. We describe some lessons learned and decisions made in the process of reconstructing high-level user actions in Alice from low-level Alice logs.

Categories and Subject Descriptors
K.3.1 [Computers and Education]: Computer Uses in Education – *Computer-managed instruction.* K.3.2 [Computers and Education]: Computer and Information Science Education – *Computer science education.*

General Terms
Human Factors, Languages.

Keywords
Educational data mining, Education, Middle school, Alice.

1. INTRODUCTION
Computer logging data has great potential for informing research and tool development in education due to the low-cost ability to track every action with today's computer systems. For many years, computer scientists have been using data analytics techniques with web usage, computer interaction, and game play logs in order to inform tool development and to predict student performance [4]. However, few studies have looked at what can be learned from this data in the middle school context. This paper describes the first steps in the process of using low-level logging data of middle school students using Alice, a 3D environment using drag-and-drop programming [8]. We describe the steps that were required and problems encountered in order to prepare the logging data to be used to describe how students approach two different tasks: programming games (an open-ended task) and a performance assessment (a closed-ended task).

Ultimately, we plan to use the logging data to answer a number of questions such as: Can logging data reveal the conditions under which students notice and use key features of an initial programming environment? Do students recognize problem patterns and try creative solutions or do they keep repeating the same ineffective strategies? Do students' strategies vary, depending on whether they are working on a closed-ended or open-ended task? Are student actions related to the sophistication of the games they produce? We hope to distinguish the development of routine expertise, where learners develop a core set of skills that they use with increasing efficiency, from adaptive expertise, which involves a balance between innovation and efficiency where learners develop meaningful knowledge so they can adapt their skills in response to new situations [1].

Before we can get very far into the process of using Alice logging data to understand students' computer programming and problem solving strategies, we need to understand the internal structure of the logs and how that maps to high-level actions of students using Alice. We also need to make a series of decisions that determine how the data will be parsed. This paper describes the problems we have encountered and discoveries we have made, starting with creating and using an Alice log parser, and then applying it to logs created during both open-ended and closed-ended tasks.

We created the parser to combine low-level individual log entries into high-level Alice actions (HLAAs). The description of this process is important for researchers and educators that want to use Alice or similar tools that generate logging data to understand how students learn to program and if and how students develop adaptive expertise. Additionally, this work is important for researchers and tool builders involved with the design of activity logging systems for other programming environments and software tools.

2. Previous Work
There is a long-standing interest in novice programming, which forms the basis for our analysis of Alice logs. Publications in the 1980s described how children develop programming skills and ways of thinking [19][20]. Studies of higher education have used phenomenology to analyze interviews with students in an introductory Java programming course to reveal how they experience the act of learning to program [4]. In addition, researchers have studied the strategies used by expert programmers hoping to use this information to teach novices more effective programming strategies [17].

Educational data mining techniques have been used to build models of student behavior when using traditional intelligent tutoring systems. Baker [3] has successfully developed models transferable to different tasks for structured problem solving

SIGCSE'2013, March 6–9, 2013, Denver, CO, USA.
Copyright 2013 ACM 978-1-4503-1868-6/13/03 …$15.00.

activities where there are designated correct responses. Using time-consuming (manual) identification of activity by domain experts, progress has been made with understanding and modeling students' exploratory use of interactive simulation systems where there is no clear identification of correct use [15]. More recent work [2] describes a data mining approach used to identify behavioral patterns that then require human labeling in order to model students' use of these unstructured learning environments. For example, researchers create a data point for each student consisting of the frequency of use of every tool action and the mean and standard deviation of the latency between actions representing reflection time. Additionally, researchers collect student pre- and post-test domain knowledge.

Logging data has incredible untapped potential for adding to our understanding of how children learn with technology, especially the class of unstructured learning environments called initial programming environments. The developers of these environments intend for the learners to "be interested to explore, and to represent their explorations" [9]. In one groundbreaking study [16]; researchers used machine-learning techniques to analyze approximately 200,000 logs of code snapshots from beginning university-level programming students using Karel the Robot, an initial programming environment. Their results are that the patterns discovered using machine-learning techniques were more predictive for students' performance on midterm exams than the grades they received on the submitted program solutions. One implication of this is that the process of programming was more predictive than the program produced. In addition, the researchers were able to show applicability of their methodology for use with Java code snapshot logs of beginning university-level programming students.

Our study focuses on Alice, a widely used initial programming environment [21], and one of few for which logs can be captured [12]. Kelleher built an Alice logging system and distinguished a log entry as representing one of three different categories of Alice activities: programming, scene layout, or program play. She found time-based usage patterns when doing open-ended tasks to be different for 12-year old girls using two different variants of Alice: Storytelling Alice or Generic Alice (without storytelling support) over a 4-hour period. Identification of these high-level actions is an informative first step in mining the Alice logs. However, despite the use of Alice in many K-12 and college classrooms around the world (there are over 23,500 members in the Alice online community forum), there is no existing system for capitalizing on these logs to understand novices and how they learn to program [7].

Our log data come from 320 middle school students who were enrolled in voluntary after-school or elective during-school technology courses called iGame. These students ranged in age from 10-14 years (mean=12) and they had, on average, 2.7 computers at home. Students were primarily white (46%) or Latino/a (37%). Sixty-three percent of the students were male. Out of the 134 students who said they spoke another language at home at least some of the time, 108 (81%) of these students named Spanish or Spanish and another language.

The research we describe here will contribute to the growing number of efforts to use educational data mining techniques, and especially learning analytics to describe how students interact with computers and the implications for learning. Eventually, this knowledge can be used to inform strategies for teaching and assessing the learning of novice programmers, as well as for improving programming environments. Specifically, it will help us understand how novice programmers develop adaptive expertise.

3. Ana2, an Alice Log Parser

Based on Kelleher's description of her research [13], with a few exceptions, she places each individual log entry into one of four categories (programming, layout, play, or ignore) according to the value of the event field of the log entries. It is our desire to understand more about what the user is doing than to determine the amount of time spent doing programming, layout, or play, We want to be able to identify individual high-level user actions. It became clear that looking only at individual low-level log entries was insufficient for identifying high-level actions because most user actions result in sequences of log entries. We have found that combining sequences of one or more low-level log entries into High-Level Alice Actions (HLAAs) is necessary to create the 'primitive pieces' that mirror the user actions. These primitive pieces can then be organized into sequences that are instances of specific problem solving strategies. To this end, we built a parser, Ana2, based on an earlier attempt [18], to combine the log entries into HLAAs. In this section we discuss our approaches to answering a number of questions that arose during the process of building Ana2 and reasons for the choices we made.

3.1 How to combine log entries into HLAAs.

Our efforts to combine log entries into HLAAs were challenged by the lack of formal documentation for the Alice logging system. The source code is public, but it contains essentially no comments. We learned the format of the log entries from various methods [6]7,13,[18]] including the use of reverse-engineering techniques.

We learned that every log file is composed of records called entries; one or more entries are written for each HLAA. For example, inserting a mouse click event handler, generates ten low-level log entries. Each log entry has two required fields and a variable number of optional fields, each with field values. The required fields are time (in milliseconds) and event (with 12 possible values). The number and type of other fields in each log entry are dependent on the value of the event field. Most log entries have many fields and many of the values of the fields are dependent on objects, operations, and other pieces of the story narrative the student chooses to model with their Alice program.

The first version of the Alice log parser, Ana, relied entirely on the time gaps between log entries in order to combine the log entries into HLAAs. Ana used a total accumulated time threshold of 800 milliseconds (ms). This meant that an HLAA was composed of a log entry and all subsequent log entries, provided the total elapsed time was less than 800ms [18].

The threshold of 800ms was somewhat arbitrary and we found that sometimes multiple HLAAs could be found in a sequence of log entries with an elapsed time less than 800ms (e.g. multiple quick mouse clicks to adjust the position of an object would each be a single layout HLAA). Furthermore, we also discovered examples of single HLAAs that spanned more than 800ms (e.g. using Storytelling Alice and switching to a different scene with many objects). In our first attempt to refine this, we switched to using the time gap between two consecutive log entries, rather than the total elapsed time accumulated for the HLAA to decide when to start a new HLAA. If the time gap between two consecutive log entries exceeded a certain value, which we will call the 'maximum log entry gap' (MLEG), then a new HLAA was assumed to start with the log entry following that gap. This seemed to do a better job of separating low-level log entries into HLAAs corresponding to single user actions, but we were still unable to find an appropriate value for the MLEG (on the order of 200-500ms) that satisfactorily separated the sequences of entries into HLAAs.

Some HLAAs can be identified regardless of the time gaps between log entries. Unfortunately, it is not possible to unambiguously reconstruct all of the individual HLAAs from the logs. Our current version of the Alice log parser, Ana2, uses a combination MLEG with a grammar for the types of HLAAs we expect to see in the logs. For example, the sequence of log entries shown in Figure 1 has a 281ms gap between the timestamps for the 2nd and 3rd entries. This could be interpreted as two HLAAs, the recording of a sound (the first two entries) followed by the addition of a play sound method call (the last two entries), or as a single insertion of a play sound method call, selecting the "record new sound" option. The only way to decide between these two choices is based on the time gap between the entries having event field values of objectArrayInsert and insertResponse, which has been observed to be anything from 0 milliseconds to many seconds in the log files we've analyzed. If the time gap is many seconds, then this sequence is clearly two HLAAs; if the time gap is zero or just a few milliseconds, then clearly it is one HLAA. This means we are left making a judgment call about the size of the time gap, if it alone is used to separate two or more HLAAs. Fortunately, for some analyses, the choice between one or two HLAAs is irrelevant provided we categorize both HLAAs the same way (as programming HLAAs in this case). However, we can envision situations where it might matter. For example, we might decide that recording of sounds is more like scene layout and should be included in the layout category rather than the programming category. If it were seen as one HLAA, it would be a programming HLAA. If it were seen as two HLAAs, it would be a layout HLAA followed by a programming HLAA.

```
Time=1…277921,Event=insertChild,childtype=<…Sound>
Time=1…277937,Event=objectArrayInsert,…,<sounds>
Time=1…278218,Event=insertResponse,…<SoundResponse>
Time=1…278234,Event=objectArrayInsert
```

Figure 1: Example of an event sequence yielding one or two HLAAs depending upon the inter-log-entry gap which is 281ms between the 2nd and 3rd events in this sequence.

3.2 How sensitive are our initial key metrics to possible errors in our combining of log entries into HLAAs?

Any choice of MLEG is likely to result in either some high-level user actions being split into two HLAAs because of a large time gap, or two high-level user actions being combined into one HLAA because of a small time gap between the end of the first high-level user action and the start of the next. In this section we examine how sensitive our initial metrics are to our choice of MLEG and discuss our search for an MLEG that would affect the fewest number of HLAAs.

Using an MLEG of zero, there is roughly a uniform distribution of gap times preceding a layout HLAA in the range 300ms to 999ms. For programming HLAAs, the distribution of gap times in this same range is small enough to be of little or no consequence with less than 0.3% of the log entries separated by gaps in the range 200-1000ms. Therefore, the identification of programming HLAAs is relatively insensitive to the choice of an MLEG anywhere in this range.

Based on our manual inspection of the logs, using an MLEG greater than 300ms will result in combining HLAAs that were quite likely from two separate, but closely spaced, high-level user actions. Choosing an MLEG less than 300ms results in splitting many potential HLAAs with small inter-event gap times into two separate HLAAs. In all of the logs we processed moving from a 300ms MLEG to a 200ms MLEG resulted in an 1.4% increase in the number of HLAAs, however, we have not seen clear evidence of log entry sequences that should be split using these small gap times (below 300ms). The identification, via manual inspection, of HLAAs that are separated by approximately 300ms combined with the small increase in total HLAA count when moving below 300ms has led us to use a value of 300ms for the MLEG. Thus, we programmed the parser to split HLAAs when the inter-log-entry gap exceeded an MLEG of 300ms, unless it was overruled by a grammar rule that unambiguously separates or combines the two adjacent log entries (e.g. a stop event immediately followed by a save event which would always be separated).

A key metric reported by Kelleher [12] is the ratio of programming time to layout time. Kelleher used this metric to determine whether students spent most of their time doing layout, programming, or a balance of the two. Table 1 shows this metric, total programming time (computed as the sum of the total gap times prior to all programming HLAAs and the total elapsed times of those programming HLAAs) divided by total layout time (computed as the sum of the total gap times prior to all layout HLAAs and the total elapsed times of those layout HLAAs) and two additional key metrics: average gap time prior to a programming HLAA, and average gap time prior to a layout HLAA. As shown in Table 1, the results are relatively insensitive to an MLEG choice anywhere in the 100-400ms range. The ratio of total programming time to total time doing layout is unchanged when rounded to four significant digits. In addition, in the 200-400ms range, the average gap time prior to a programming HLAA varies by less than one-tenth of a second and the average gap time prior to a layout HLAA varies by less than one-half of a second. The larger change for the average gap time prior to a layout HLAA is expected because this is where we see many individual high-level user actions in close succession (minor adjustments to the position of a character) that are indistinguishable, based on our manual analysis, from compound layout actions triggered by a single high-level user action. When identification is ambiguous using our knowledge of Alice and its logging system, our use of an MLEG of approximately one-third second (i.e., 300ms) allows us to identify quick mouse clicks while doing scene layout as individual layout HLAAs (provided the user pauses for at least 300ms). Yet it correctly identifies as a single layout HLAA, many long sequences of log entries generated when students use the right mouse button to execute a scene layout action.

gap time->	100ms	200ms	300ms	400ms
programming time / layout time	1.627	1.627	1.627	1.627
average programming gap time (secs)	28.30	28.54	28.58	28.60
average layout gap time (secs)	10.43	10.56	10.84	10.96

Table 1: Key **metrics to test gap sensitivity.**

In a few specific situations, primarily related to the initial loading of a scene done for a program play HLAA or related to a scene layout HLAA when done using the right mouse button, we allow even larger inter-log-entry times, up to 5 seconds for log entries within a single HLAA. These gaps always precede a log entry with 'propertyChange' as the value of the event field. In the log files that we analyzed, consisting of over 800,000 log entries combined into over 200,000 HLAAs, we found only 96 instances where this long gap exception occurred.

3.3 How should the time between HLAAs be treated?

During the time interval between HLAAs, the student may think about or reflect on what was just done, what to do next, or some combination of both of these. We call this time "reflection time." When the two bordering HLAAs are both in the same category of either programming or scene layout, then it is reasonable to conclude that the time was spent reflecting about programming or scene layout, respectively. However, when the two bordering HLAAs are from different categories, was the student reflecting on what they just did, or was the student reflecting about what to do next? Of course, it is also possible in either situation that the time between HLAAs could be due to other actions, such as taking a break, or waiting for help from a teacher. When the HLAAs are from different categories, we associate the reflection time with the second HLAA; that is, viewing the time primarily as reflecting about what to do next. For example, the time between a programming HLAA until the start of the following scene layout HLAA is associated with the scene layout HLAA.

An exception to this reflection time allocation involves the 'save' HLAA. Typically students using Alice save their work for three different reasons: because they have completed a series of changes they don't want to lose, because a 'save' prompt appears, or because they are ready to quit their work with Alice. If a student does a save and then quits, we allocate the time leading up to the save as reflection time for the save. If a student does a save and then continues to do more work with Alice, then the time leading up to the one or more saves and the time following the save until the next non-save HLAA is allocated as reflection time for this following HLAA. The reasoning for this becomes clear when one considers the scenario of saving because a 'save' prompt appears. During this scenario, the student is thinking about the next action; is interrupted by the 'save' prompt; saves; then continues with thoughts about the next action. Clearly, all of this reflection should be allocated to thinking about the action following the save HLAA.

3.4 Does the identification of HLAAs distinguish between programming behavior in closed versus open-ended tasks?

As a first-check to see if our identification of HLAAs and the categories adopted from Kelleher are going to be useful for understanding behavior in initial programming environments, we compared our key metrics for two different activities. The first was an open-ended task where students programmed a computer game, either alone or with a partner. The second was a close-ended task of an assessment of computational thinking that we call the Fairy Assessment (FA) done by each student individually. The FA was built in the style of a computer game and students were asked by the dialog of the characters within the FA to solve three subtasks [22].

As shown in Table 2, the metrics are very different for students using Alice when doing the FA versus open-ended game programming. Since we expected students to do little, if any, layout when doing the FA, we are not surprised at the ratio of programming time to layout time for the FA and this finding serves as a validation of our process.

	FA	Games
Ratio of programming time/layout time	8.512	1.435
average programming think time (secs)	25.83	29.31
average layout time (secs)	12.86	11.13

Table 2: Comparing key metrics for FA vs. game programming.

Table 3 shows a summary of our findings for percentages of time spent in layout, programming, and program play, and compares them to the findings of Kelleher [12]. Kelleher's data are from a study of 88 middle school girls using either Generic Alice (GA in the table) or Storytelling Alice (SA). Because Kelleher's time calculation is based on timestamps between log entries; in order to make a comparison of Kelleher's results to ours, we combine the action times with the reflection times for these same HLAA categories. The fact that our values are similar to Kelleher's is an additional validation of our analysis process.

	Ana2	Kelleher GA	Kelleher SA
Programming	36.8%	34.0%	48.3%
Layout	22.7%	40.8%	22.3%
Program play	34.9%	25.1%	29.3%
Other	5.6%	0.1%	0.1%

Table 3: Distribution of time by category from our analysis and that of Kelleher [12].

4. Discussion

This work has implications for tool developers implementing logging systems for programming environments and other software tools. Here we make recommendations for logging systems. They are:

1) Integrate the logging into the design of the software. Researchers looking at introductory programming trace logs from university students have found that the process of program creation is a better predictor of student test scores than the finished program [16]. Logs can give us information about the process of program creation but it is not always easy to identify HLAAs from sequences of low-level log entries. If the programming environment software is designed and built with high-level action logging, then an entry could be logged for each high-level action and each high-level action could have a corresponding log entry, or easily identifiable sequence of log entries (e.g. beginAction … endAction).

2) Include a 'start' log entry when the software opens. We don't know how much reflection time is present before the current first log entry in each file. Currently, the first HLAA in each file has a zero reflection time.

3) Similarly, include a special log entry when a new file/project is opened, or when a new project is created. These are critical boundaries that are obscured in the current Alice logging system.

4) It appears important to capture time-off-task. Certainly eye-tracking hardware can be used to determine if a student is physically present. Alternatively, if privacy and cost are concerns, another strategy that can be used to determine a student's presence is to fade out the display after a time of keyboard or mouse inactivity. Provide a mechanism to set the inactivity time prior to fade out and to disallow student modification of it. When the student makes a keystroke, a log entry can be made to record the

time from the fade out until the keystroke. This will not capture reflection time during which a student goes to another student or the teacher for help, however, it does correctly identify other time-off-task.

5) Assist in the design and development of an open data repository for logs from novice programming environments. Alice has a logging system that needs modification to make it more easily usable for the educational data mining community using the recommendations we give here. BlueJ has a logging system that has been used to study novice compilation behavior [10][11]. Other programming environments designed for novices such as Scratch, Agentsheets, Stagecast Creator, and Greenfoot would benefit from logging features. Alternatively, design the logging data to be consistent with the standards being developed by the PSLC DataShop group [14]. DataShop is an open data repository of logs from intelligent tutoring systems designed for the educational data mining community.

6) The interface to the Alice programming environment has an undo/redo function. This provides an undo mechanism to the Alice user so that any HLAA back to the start of an Alice session can be undone. Similarly, HLAAs can be redone if they have been undone. Unfortunately, the Alice logging system does not capture any redo/undo actions; we have no record of any redo or undo actions a student has done. Future logging systems should ensure that these important high-level user actions are captured in the log.

7) Consider the important questions to be asked from analysis of the logs before the design of the logging system is complete. The answers to these questions should give us a starting place. For us, we want to determine how novices use the Alice software. We want to determine what constitutes a 'problem solving strategy' and if novices use different problem solving strategies. Researchers have used terms like tinkering and flailing. To see flailing in logs, the use of undo and redo needs to be captured.

5. Conclusions

The study described in this paper adds to the body of literature in learning analytics in two ways:

1) We started the important work of analyzing the logs of young students using initial programming environments, specifically Alice, for the purposes of learning how they use these programming environments when working on open and closed-ended tasks. Eventually, this knowledge can be used to inform strategies for teaching and assessing the learning of novice programmers as well as for improving programming environments.

2) We identified problems and decision-making steps for the analysis of Alice logs so that researchers and tool developers can improve the logging features of other initial programming environments and software tools to facilitate learning analytics.

We have built a parser, Ana2, and used it to identify HLAAs from the logs. The report of our mean time values reaffirms the times reported by Kelleher [12] and prepares us to go further with our plans to use unsupervised and supervised classification techniques similar to those described by Amershi and Conti [2], to address questions such as the following:

1) Can Alice logs be used to distinguish between programming strategies (e.g. linear development based on a plan vs. central algorithm followed by initializations) used by novices when programming games?

2) Can Alice logs be used to distinguish between problem solving strategies (e.g. flailing vs. tinkering) used by novices when working on the FA assessment?

3) Can Alice logs be used to distinguish between programming strategies used by students pair programming compared to strategies used by students working solo when programming games?

4) Can Alice logs be used to determine if there are differences in the findings for the closed versus open-ended tasks? If so, what do these differences mean?

We now have the data we need (logging information) in the form we need (High-Level Alice Actions) to begin to look for patterns and programming strategies that will help us answer the questions we posed in the Introduction, such as "Do students recognize problem patterns and try creative solutions or do they keep repeating the same ineffective strategies?"

6. ACKNOWLEDGMENTS

Thanks go to Shannon Campe and Eloy Ortiz, whose vital support was necessary to run the classes and collect logging data. This research is partially funded by a grant from NSF 0909733 "The Development of Computational Thinking among Middle School Students Creating Computer Games." Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] Alexander, P. 2003. The development of expertise: The journey from acclimation to proficiency. *Educational Researcher*, 32, 10-14.

[2] Amershi, S., and Conati, C. 2009. Combining unsupervised and supervised classification to build user models for exploratory learning environments. *The Journal of Educational Data Mining*, 1(1), 18-71.

[3] Baker, R.S.J.d. 2007. Modeling and understanding students' off-task behavior in intelligent tutoring systems, *Proceedings of ACM CHI 2007: Computer-Human Interaction*, 1059-1068.

[4] Baker, R., & Yacef, K. (2009). The state of educational data mining in 2009: A review and future visions. *Journal of Educational Data Mining, 1*(1), 3-17.

[5] Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., and Stoodley, I. 2004. Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education*, 3, 143-160.

[6] Conway, M.J. 1997. Alice: Easy-to-learn 3D scripting for novices. University of Virginia, Ph.D. Dissertation.

[7] Cooper, S. 2010. Personal conversation. January 24, 2010.

[8] Dann, W., Cooper, S., and Ericson, B. 2009. *Exploring Wonderland: Java Programming Using Alice and Media Computation.* Prentice Hall.

[9] Fincher, S., and Utting, I. 2010. Machines for thinking. *ACM Transactions on Computing Education*, 10(4).

[10] Jadud, M.C. 2005. A 1st look at novice compilation behavior. *Computer Science Education*, 15(1), 25-40.

[11] Jadud, M.C., and Henriksen, P. 2009. Flexible, reusable tools for studying novice programmers. *ICER'09*, August 10-11, 2009, Berkeley, CA. USA.

[12] Kelleher, C. 2006. Motivating programming: Using storytelling to make computer programming attractive to more middle school girls. Carnegie Mellon University Ph.D. Dissertation, September 2006.

[13] Kelleher, C. 2008. Emails July 14, 2008.

[14] Koedinger, K.R., Baker, R.S.J.d., Cunningham, K., Skogsholm, A., Leber, B., Stamper, J. 2010. A data repository for the EDM community: The PSLC DataShop.

[15] Merten, C., and Conati, C. 2006. Eye-tracking to model and adapt to user meta-cognition in intelligent learning environments. In *Proceedings of the 11th International Conference on Intelligent User Interfaces (IUI '06)* ACM, New York, NY, USA, 39-46.

[16] Piech, C., Cooper, S., Sahami, M., Koller, D., and Blikstein, P. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, New York, NY, USA.

[17] Robins, A., Rountree, J., and Rountree, N. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137-172.

[18] Smith, B. 2010. Effect of pair programming on middle schoolers using Storytelling Alice. *13th Annual UCSC Undergraduate Research Symposium*.

[19] Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. 1983. What do novices know about programming? In B. Shneiderman & A. Badre (Eds.) *Directions in Human-Computer Interactions,* 27-54. Norwood NJ: Ablex.

[20] Soloway, E., and Spohrer, J.C. (Eds.) 1989. *Studying the Novice Programmer.* Hillsdale N.J.: Lawrence Erlbaum.

[21] Utting, I., Cooper, S., Kolling, M., Maloney, J., and Resnick, M. 2010. Alice, Greenfoot, and Scratch – A discussion. *ACM Transactions on Computing Education*, 10(4), 1-11.

[22] Werner, L., Denner, J., Campe, S., Kawamoto, D.C. 2012. The Fairy Performance Assessment: Measuring computational thinking in middle school. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE),* ACM, New York, NY, USA.