A First Step in Learning Analytics: Pre-processing Low-Level Alice Logging Data of Middle School Students

LINDA WERNER
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA USA

CHARLIE MCDOWELL
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA USA

AND

JILL DENNER
Education, Training, Research Associates
Scotts Valley, CA USA

Educational data mining can miss or misidentify key findings about student learning without a transparent process of analyzing the data. This paper describes the first steps in the process of using low-level logging data to understand how middle school students used Alice, an initial programming environment. We describe the steps that were required and the decisions that were made in building a tool to translate the low-level logging data into a form that can be used to investigate educational questions about problem solving strategies for a range of different programming tasks. This work contributes to efforts to analyze educational data, and is important for researchers and tool builders involved with the design of logging systems for other programming environments and software tools.

---

1 INTRODUCTION

The field of educational data mining is growing (Baker & Yacef, 2009), but to date, few teams have tried to make logging data directly useful to educators (Macfadyen & Dawson, 2010). Logging data is computer-generated data that tracks actions made by the user; the exact content depends on how the logging data system is designed. There is great potential for these data to inform how students learn and problem solve while producing technology such as creating computer games, if the analysis is driven by both theoretical and empirical approaches. To this end, the field would benefit from a careful description of the steps required and the types of decisions that will need to be made in order to translate large amounts of data into meaningful information.

This paper describes the first steps in the process of analyzing low-level logging data of middle school students using Alice, a 3D environment using drag-and-drop

programming, a language that is closely related to Java, and many other modern imperative programming languages (Dann et al, 2009). We describe the steps that were required and problems encountered in building a tool to translate the low-level logging data into high-level actions that can be used to investigate educational questions about problem solving strategies for a range of different programming tasks. We specifically are interested in developing a transferable process that others can use to understand how students approach different tasks: open-ended (e.g., programming games) and closed-ended (e.g., completing a performance assessment).

In this paper we describe the first steps to prepare the data to address several types of research questions, such as: Can logging data reveal the conditions under which students notice and use key features of Alice? Do students recognize problem patterns and try creative solutions or do they keep repeating the same ineffective strategies when problem solving in Alice? Do students' strategies vary, depending on whether they are working on a closed-ended or open-ended Alice task? Are student actions related to the sophistication of the Alice games they produce? In particular, we are exploring ways of leveraging logging data to create new models of *adaptive expertise* by drawing on multiple disciplines: computer science (specifically computer science education, machine learning, data analytics, and programming), the learning sciences, and psychology. Hatano (1988) describes adaptive expertise as the meaningful and well-connected knowledge that can be applied to new tasks. To our knowledge, no other studies have used logs 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 (Alexander, 2003). Most assessments are limited by their focus on how students develop routine expertise, so there is a need for strategies to assess how students try out ideas, when and how they receive feedback, and how they respond to that feedback (Bransford et al., 2006).

The focus of the current article is to describe the first step in how to use the logs to understand learning. Before Alice logging data can be used to understand students' computer programming and problem solving strategies, it is necessary to understand the internal structure of the logs and how that maps to higher level actions of students using Alice and to make a series of decisions that determine how the logs will be parsed. This paper describes the problems we have encountered and discoveries we have made, starting with creating an Alice log parser, and then applying it to logs created during both open-ended and closed-ended tasks. The parser is automated to combine low-level

individual log entries into higher level actions that we call high-level Alice actions (HLAAs) each corresponding to functionality provided at the user interface level of the Alice programming environment.

The description of our process is especially important for researchers who work with systems whose logging subsystems record only low-level actions, and we describe how parts of our process are applicable to logging systems in many other domains. Additionally, this work is important for researchers of Alice, who can also use our Alice log parser to pre-process Alice logging data generated in a range of settings, and analyze it to understand how students learn to program. Finally, this work is relevant to tool builders involved with the design of activity logging systems for other programming environments and software tools, and we include recommendations based on our experience with building our Alice log parser.

2 PREVIOUS WORK IN LEARNING ANALYTICS AND APPLICATION OF THIS TO ALICE LOGS

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 (Soloway et al., 1983; Soloway & Spohrer, 1989). 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 (Bruce et al., 2004). In addition, researchers have studied the strategies used by expert programmers with the hope that this information can be used to teach novices more effective strategies (Robins et al., 2003). Robins et al. write, "Experts can typically retrieve relevant plans from memory, and then generate code from the plan in linear order (from initialisation, to calculation, to output). Novices must typically create plans. ….Code generation begins with the central calculation, and builds the initialisations and other elements around it." (p. 144). However, most of this research has relied on observations and interviews with small samples, which are time consuming and have limitations in their precision and generalizability. These data are neither detailed enough nor extensive enough to distinguish between paths to different types of expertise.

Educational data mining techniques have been useful to build models of student behavior when using traditional intelligent tutoring systems. Baker (2007) has successfully developed models transferable to different tasks for structured problem solving activities where there are designated correct responses. Dimitracopoulou (2005) describes work using traces of expected interactions to guide collaborations. In addition,

progress has been made with understanding and modeling students' exploratory use of interactive simulation systems where there is no clear identification of correct use. Initial approaches involved time-consuming (manual) identification of activity by domain experts (Merten & Conti, 2007). More recent work (Amershi & Conti, 2009) 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, they 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, they collect student pre- and post-test domain knowledge.

Logging data has incredible untapped potential for adding to our understanding of how children learn or fail to 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" (Fincher & Utting 2010). In work by Choquet and Iksal (2007), one case study described how characterizations of new users' exploration activities, such as sequences of use and time of use, derived from log files can be compared to 'typical exploration activities' by successful novice users of the same software. An implication of this is that it can be determined when new users are unsuccessful at exploring new software. In a groundbreaking study (Piech et al., 2012); researchers used machine-learning techniques to analyze approximately 200,000 logs of code snapshots from beginning university-level 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 solution. One implication of this is that the process of programming was more predictive than the end result. 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 (Utting et al., 2010), and one of few for which logs can be captured (Kelleher, 2006). Kelleher built an Alice logging system and distinguished a log entry as belonging to 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

categories is an informative first step in analyzing the Alice logs. However, despite the use of Alice in many K-12 and college classrooms across the US, 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 computer program (S. Cooper, personal conversation, January 24, 2011).

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, we hope it will help us understand how novice programmers develop adaptive expertise.

3 RESEARCH QUESTIONS

Building on prior research described above, our first investigative step was to create a parser for these Alice logs, use it to translate the low-level logs into logs containing high-level user actions or HLAAs, categorize these HLAAs into categories similar to those Kelleher used in her Alice logging work, and determine whether the distribution of HLAAs into these categories varies across programming tasks. Our research questions were:

- Can Alice logs be translated into logs of high-level Alice actions (HLAAs)?
- How do we create a parser to do this?
- Can we place the HLAAs into one of three categories (programming, scene layout, and program play) in a way that agrees with the findings of Kelleher?
- Can we determine the time a programmer spends in each of these categories?
- Do student actions as represented by characterizations of the HLAAs vary across different types of programming tasks?

4 OUR ANALYTIC APPROACH

4.1 Logging Data

Our data were collected as part of a study of the development of computational thinking (CT) among middle school students programming computer games. We define CT as the application of computer science concepts to identify and solve problems by creating a model (e.g., a game), effectively automating ideas, and transferring that knowledge to new situations (e.g., the assessment).

Students used two variants of the Alice programming environment: Storytelling Alice (SA) and Alice 2.2. See Figure 1 for a screen shot from the Alice 2.2 programming environment containing a scene from one of our student's games. We show this to give readers an idea of the learning environment and the complexity of the user interface. The user interface has multiple panels. Starting from the top left corner and moving clockwise, there is: the object tree showing all objects currently in the program, the scene editor, the event handler editor used to program for user interaction, the program code editor, and details about available operations and characteristics for each of the program's objects. Additionally, across the top there are buttons for playing the program, for undo and redo of user actions, and file operations to open existing Alice programs and to create new ones. In the rest of this paper, we will use Alice to mean both Storytelling Alice and Alice 2.2 unless it is important to differentiate.

We have logs collected from students working on two tasks. The first is an open-ended task where students program a computer game, either alone or with a partner. Students spent approximately 10 hours making their game, and were allowed to make any type of game, with the only limitation that it be appropriate for school. The second is a close-ended task that we call the Fairy Assessment (FA) done by each student individually. The FA was built in the style of an Alice computer game and students are asked to complete three subtasks.

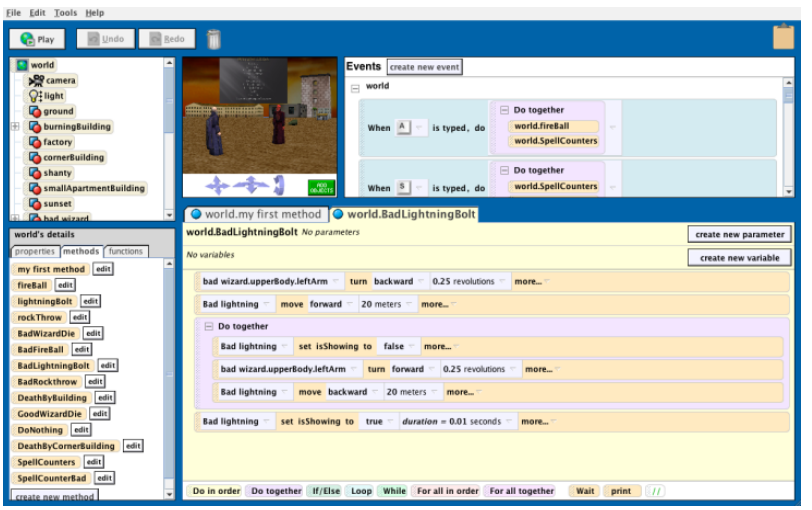

Figure 1: Student-created 'Wizard Duel' game.

Instructions for the FA's three assessment tasks come from character dialogue and a list at the bottom of the play window. The analysis of students' performance on the FA is described in an earlier paper (Werner et al., 2012). The results suggest that the

assessment was successful in picking up a range of CT across students and a variety of types of CT.

4.2 PARTICIPANTS

The data come from 320 middle school students who were enrolled in voluntary after-school or elective (during-school) technology courses. The classes were randomly assigned to a pair programming (McDowell et al., 2006) or solo programming condition. 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%). 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. Table 1 includes key descriptives on all participants that completed a post-course survey, and shows that there were more male than female students, more enrolled in during-school electives than after-school classes, and there was a range of mother's education.

Table 1: Key Descriptives on Participants (n = 320).

By Gender	Total
Female	118
Male	202
Totals: By Gender	320
By Setting	
During School	198
After School	122
Totals: By Setting	320
Mothers' Education	
1: Less than high school	33
2: Completed high school	37
3: Some post-secondary (w/o community college or university degree)	97
4: University degree with or without post grad degree	102
Totals: mothers' education	269
Mean mother's education (standard deviation) N=269	3(1.006)

4.3 ANALYSIS PROCESS

Our effort to combine the low-level log entries into high-level Alice actions (HLAAs) extends the work by Kelleher in several ways. Kelleher dealt only with individual log entries (C. Kelleher, personal communication, July 14, 2008) but we have found that combining sequences of one or more low-level log entries into HLAAs is necessary to create the 'pieces' that mirror the user actions and that correspond to functionality provided at the user interface level of the Alice programming environment. Sequences of these HLAAs can be recognized as specific problem solving strategies. We built a parser, Ana2, to translate the logs of low-level log entries into logs of HLAAs, based on an earlier version called Ana, written by then UCSC undergraduate student Ben Smith (Smith, 2010).

Following are some of the questions that guided the design, and building of Ana2, and our interpretation of the data it generates. In the rest of this paper, we discuss our approaches to answering these questions, the reasons for the choices we made, and the lessons learned for the analysis and interpretation of other logging data systems.

- Can a single log entry be used to identify an HLAA?
- How can log entries be translated into HLAAs?
- How sensitive are the initial key metrics to possible errors in the translation of log entries into HLAAs?
- Into what category (programming, layout, play) of HLAAs do we allocate scene changes?
- How should the time between HLAAs be treated?
- How important is an HLAA's elapsed time (i.e., the time from the start of an HLAA until the last timestamp on log entries still designated as part of that HLAA) and how should it be treated?
- How can we determine whether gap times are due to the student reflecting or to the student being off-task?
- Does the identification of HLAAs distinguish between programming behavior in closed versus open-ended tasks?

4.4 CAN A SINGLE LOG ENTRY BE USED TO IDENTIFY AN HLAA?

Based on Kelleher's description of her research (C. Kelleher, personal communication, July 14, 2008), each individual log entry is ignored or assigned to one of three categories (programming, layout, or play) according to the value of the EVENT field of the log entries, with a few exceptions that also look at values of other fields. In our efforts to understand the structure of the Alice logs, it became clear that individual low-level log entries were insufficient for identifying HLAAs. This is because some user actions each result in sequences of log entries that include a combination of Kelleher's three different categories: programming, layout, or play. For example, the programming HLAA (INSERT_EVENT) we have identified in Figure 2, which results from inserting a mouse click event handler into the program, contains low-level log entries that would fall into the programming category and other low-level log entries that would fall into the layout category based on Kelleher's categorization (see Table 2).

Table 2: Kelleher's categorization of low-level log entries (C. Kelleher, personal communication, July 14, 2008). Log entries without these EVENT field values are ignored.

Log entry EVENT field	Exceptions	Category
oneshot		layout
insertChild		layout
deleteChild		layout
insertResponse		program
deleteResponse		program
insertCallToUserResponse		program
deleteCallToUserResponse		program
propertyChange	not a localTransformation or vehicle change	program
objectArrayInsert	mentions response	program
objectArrayDelete	mentions response	program

4.5 HOW CAN LOG ENTRIES BE TRANSLATED INTO HLAAS?

Our efforts to separate 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 original documentation about the creation of Alice (Conway, 1997), from research results by a member of our team (Smith, 2010), from conversations with Alice programmers and

researchers (C. Kelleher, personal communication, July 14, 2008, S. Cooper, personal communication, January 24, 2011), and from the use of reverse-engineering techniques.

```
…
TIME=<1262899238818> EVENT=<insertChild>
      CHILDTYPE=<edu.cmu.cs.stage3.alice.core.behavior.MouseButtonClickBehavior>…
TIME=<1262899238984> EVENT=<objectArrayInsert> ELEMENT=<World.Scene 2 event2>
      OBJECTARRAYPROPERTY=<behaviors> PROPERTYOWNERTYPE=…
TIME=<1262899238986> EVENT=<insertChild> CHILDTYPE=<edu.cmu.cs.stage3.alice.core.Variable>
      CHILDKEY=<World.Scene 2 event2.x> PARENTKEY=<World.Scene 2 event2> NEWINDEX=<0>
TIME=<1262899238987> EVENT=<propertyChange> PROPERTYNAME=<valueClass>
      PROPERTYOWNER=<World.Scene 2 event2.x> OLDVALUE=<null> NEWVALUE=…
TIME=<1262899238988> EVENT=<objectArrayInsert> ELEMENT=<World.Scene 2 event2.x>
      OBJECTARRAYPROPERTY=<details> PROPERTYOWNERTYPE=…
TIME=<1262899238990> EVENT=<insertChild> CHILDTYPE=<edu.cmu.cs.stage3.alice.core.Variable>
      CHILDKEY=<World.Scene 2 event2.y> PARENTKEY=<World.Scene 2 event2> NEWINDEX=<1>
TIME=<1262899238992> EVENT=<propertyChange> PROPERTYNAME=<valueClass>
      PROPERTYOWNER=<World.Scene 2 event2.y> OLDVALUE=<null> NEWVALUE=…
TIME=<1262899238993> EVENT=<objectArrayInsert> ELEMENT=<World.Scene 2 event2.y>
      OBJECTARRAYPROPERTY=<details> PROPERTYOWNERTYPE=…
TIME=<1262899238995> EVENT=<insertChild>
      CHILDTYPE=<edu.cmu.cs.stage3.alice.core.question.PickQuestion> CHILDKEY=…
TIME=<1262899238996> EVENT=<objectArrayInsert> ELEMENT=<World.Scene 2 event2.what>
    OBJECTARRAYPROPERTY=<details> PROPERTYOWNERTYPE=…
…
```

Figure 2: Excerpt from a log corresponding to one programming HLAA – 'insert a mouse click event handler'

These contacts explained that every log file is composed of records called entries; one or more low-level entries (events) are written for each HLAA. Figure 2 shows an example of an HLAA (INSERT_EVENT in Table 3) resulting from inserting a mouse click event handler, which is composed of ten low-level log entries. As indicated by the ellipses, most of these log entries have additional fields that are not shown in the figure to save space. Each log entry has two required fields and a variable number of optional fields, each with field values. The required fields are TIME (where the field value is a timestamp in milliseconds) and EVENT (with possible field values: insertChild, deleteChild, objectArrayInsert, objectArrayDelete, insertCallToUserResponse, deleteCallToUserResponse, insertResponse, deleteResponse, oneshot, World, propertyChange, or objectArrayShift). The number and type of other fields in each log entry are dependent on the value of the EVENT field. For example, if EVENT=World, then the log entry has three fields: TIME, EVENT, and TYPE. The field value for TYPE in that case can be either start, stop, or save, corresponding to the Alice actions of play the animation, stop the animation, and a save of the Alice world to a file. Only a few of the log entries are this simple. 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 the HLAAs. Ana used a total accumulated time threshold of 800 milliseconds (ms). This means that an HLAA was composed of a log entry and all subsequent log entries provided the total elapsed time was less than 800ms (Smith, 2010). Although easy to implement, this did not produce an acceptable translation from low-level events into HLAAs.

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. switching to a different scene with many objects). In our first attempt to refine this, we used the time gap between two consecutive log entries, rather than the total elapsed time accumulated to decide when to start a new HLAA. If the time gap between two consecutive log entries exceeded a threshhold value, which we will call the maximum log entry time 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 using experimentation varying the MLEG between 200 and 500ms that satisfactorily separated the sequences of entries into HLAAs.

We were able to create a grammar (i.e., a set of rules) to unambiguously combine *some* low-level log entries into HLAAs. For a simple example, a start log event followed by a stop log event is one PLAY HLAA regardless of the time gap between the two log entries. Unfortunately, it is not possible to unambiguously reconstruct all of the individual HLAAs (listed in Table 3) from the logs using a grammar that ignores the time gaps. For example, the sequence of log entries shown in Figure 3 has a 281ms gap between the timestamps for the 2nd and 3rd entries. This could be interpreted as a sequence of two HLAAs, INSERT_SOUND followed by INSERT_METHOD, or just one: INSERT_SOUND. INSERT_SOUND can come from either the insertion of an existing sound (just the first two entries in the figure) or the insertion of a play sound method call selecting the "record new sound" option (all four entries). In the case of inserting an existing sound, the programmer may, but not always, follow this with the addition of a play sound method call (the last two entries). 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 zero milliseconds to many seconds in the log files we 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. To resolve this problem, our current version of the Alice log parser, Ana2, uses a combination of a maximum log entry time gap (MLEG) with a grammar for the types of HLAAs we expect to see in the logs.

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.

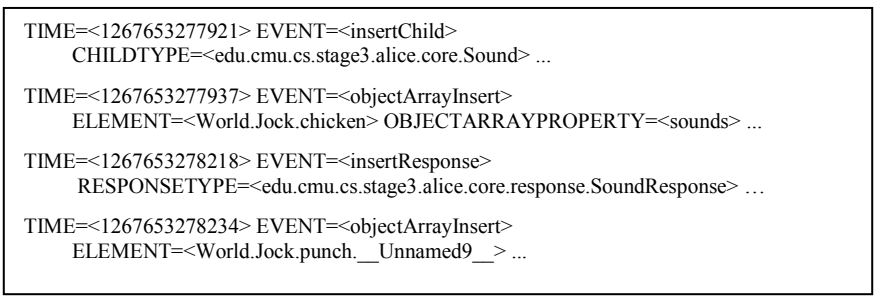

Figure 3: Events yielding one or two HLAAs depending upon the inter-log-entry gap.

We have identified 33 different HLAAs. The complete list is shown in Table 3 using names that are suggestive of the high-level user action to which they correspond. Space does not permit us to provide a detailed description of each of these HLAAs. To verify our translation process, we used the Alice 2.2 programming environment with logging enabled and created four short Alice programs, manually recording all high-level user actions as we worked. We used various layout, programming, play, and save actions. Some actions were intentionally executed quickly; others slowly. Each of these programs were animations within which a variety of objects were used and moved within each of the four scenes; we used built-in methods and created methods and event handlers. We then ran the low-level log files through Ana2 and are pleased to report that each of the four log files were translated into files with the correct sequence of HLAAs.

After an initial unsatisfactory attempt to combine low-level log entries into HLAAs using the time gap between two consecutive low-level entries, we created a grammar (set

of rules) to do the combining. We soon discovered that ignoring the time gaps altogether was also insufficient. The final parser uses a set of rules taking into account the time gap (MLEG) when required to resolve amibiguities. The resulting parser combines all low-level events from the 3122 student logs into identifiable HLAAs. Although we have no way to confirm that the HLAAs for those logs are exact matches for what the students actually did, we have also tested the parser on a small number of logs where we manually recorded the actual user actions, and in every case the HLAAs identified exactly matched the high-level user actions as expected.

Table 3: High-Level Alice Actions (HLAAs) by category.

Layout	Programming	Other
CHANGE_POSE	CHANGE_PROPERTY	PLAY_ANIMATION
CREATE_SCENE	CREATE_METHOD	PLAY
DELETE_OBJECT	DELETE_CALL	SAVE
INSERT_3D_ANIMATION_TEXT	DELETE_METHOD	
INSERT_CHARACTER	INSERT_CALL	
INSERT_MODEL	INSERT_EVENT	
INSERT_OBJECT	INSERT_METHOD	
INSERT_PERSON	INSERT_SOUND	
INSERT_POSE	INSERT_VARIABLE	
MOVE_CAMERA	INSERT_WORLD_START	
MOVE_CHANGE_VISUAL_PROP	MODIFY_ARRAY	
MOVE_OBJECT	MOVE_METHOD	
MOVE_OR_MODIFY	MOVE_METHOD_CALL	
MOVE_SUBJECT_ARROWS	OBJECT_ARRAY_SHIFT	
MOVE_SUBJECT_DRAG		
RESIZE		

4.6 HOW SENSITIVE ARE THE INITIAL KEY METRICS TO POSSIBLE ERRORS IN THE TRANSLATION 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.

Table 4 shows the distribution of gap times preceding programming HLAAs and layout HLAAs for our entire set of log files. These HLAAs are built from log entries using an MLEG of zero. That is, log entries are combined either because their timestamps are identical (i.e., they are separated by zero milliseconds) or because the grammar we have developed unambiguously identifies the sequences of log entries as specific types of HLAAs. For those sequences of log entries where we must rely on the MLEG to decide where to end one HLAA and begin the next, using a separation of one or more milliseconds results in the split of many would-be HLAAs. With an MLEG of zero, only log entries that are known to always go together, regardless of the time gap between the two consecutive log entries, are combined. For example, an HLAA that starts with an insertCallToUserResponse log entry will always be followed by either a propertyChange or objectArrayInsert log entry so they will be combined regardless of the time gap between the two.

As you can see in Table 4, for layout HLAAs formed using an MLEG of zero, there is roughly a uniform distribution of gap times between 300ms and 999 milliseconds. 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, with respect to programming HLAAs, the choice of the MLEG is not important.

Table 4: Distribution of gap times (HLAA gap times).

(ms)	0-99	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999	≥1sec.
program	28554	536	92	54	12	22	16	16	12	48	61626
layout	28350	2040	4136	1872	1564	1508	1428	1500	1328	1494	90352

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 moves us down into an area in Table 4 where we see many potential HLAAs with small gap times (4136 in the range 200-299ms) but 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 drop in the distribution for layout gap times at 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).

To test the sensitivity of using different MLEGs to further assist us in selecting an appropriate MLEG, we ran our parser generating results using MLEGs of 100ms, 200ms, 300ms, and 400ms (see Table 5). Using an MLEG of 100ms, the parser split some sequences of log entries into multiple HLAAs that were unidentifiable, meaning they did not correspond to any possible high-level user action but were part of some larger HLAA. This argues we should use an MLEG greater than 100ms. As shown in Table 4, choosing an MLEG in the range of 300-399ms is near the low end of an area of relatively uniform distribution of gap times.

Table 5: Key metrics to test gap sensitivity.

	Gap Time (MLEG)			
Key Metrics	100ms	200ms	300ms	400ms
prog time/layout time	1.627	1.627	1.627	1.627
avg prog gap time (secs)	28.30	28.54	28.58	28.60
avg layout gap time (secs)	10.43	10.56	10.84	10.96

A key metric reported by Kelleher is the ratio of programming time to layout time. This metric is useful for determining whether students spend most of their time doing layout, programming, or a balance of the two. Table 5 shows the results of our exploration of how this metric changes when we assume different MLEGs to resolve ambiguities, as discussed in the previous section. The table also includes 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 5, 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 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 correctly

identifies as a single layout HLAA many long sequences of log entries generated when using the right mouse button to execute a scene layout action.

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. We found 96 instances where this long gap exception occurred in all of the log files we analyzed consisting of a total of over 800,000 log entries combined into over 200,000 HLAAs.

In summary, at least as measured by the key metrics in Table 5, our translation from low-level log entries into HLAAs is relatively insenstive to the choice of MLEG anywhere in the range 100-400ms. Not surprisingly, we also observed that the possibility of serveral layout events in quick succession did result in some sensitivity to the choice of MLEG when examing the time gap preceeding an HLAA. This further suggests that our choice of MLEG will have a greater impact on the composition of layout HLAAs than on programming HLAAs.

4.7 INTO WHAT CATEGORY (PROGRAMMING, LAYOUT, PLAY) OF HLAAS DO WE ALLOCATE SCENE CHANGES?

Storytelling Alice allows students to easily create new scenes, place objects into scenes, and easily create code to change between scenes. However, because the logging data cannot tell us what the students' intentions were when they changed scene, it is difficult to know whether to categorize these actions into programming or layout. The problem arises when the student moves between scenes while in program development. Usually, a student moves to a scene in preparation for adding more objects to that scene. However, sometimes a student moves to a scene to determine the specific object with which to work for any programming activity that follows, even though the usual way to find the object with which to program is to use the object tree. In Alice 2.2, since the creation and movement between multiple scenes is handled differently (i.e., the student moves the camera view to another location within the single scene available, places objects at that point, etc.), movement from scene to scene is indistinguishable from other scene layout actions. Therefore, for consistency between the two Alice versions, our choice for log analysis was to place the scene movement HLAAs into the layout category.

This decision exemplifies the limitations of any analysis tool based on logs to infer the intent of the programmer. We are trying to infer two different categories of

programmer activity (programming and layout); however, the same user interaction with the tool can result from two very different types of mental processes.

4.8 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, or plan 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 a question to be answered. 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.

Although it came up during our work to verify the accuracy of our translation from low-level log events to HLAAs by examining some key metrics (see Table 5), the allocation of reflection time surrounding SAVE HLAAs is an example of the next higher level of analysis. We have identified two patterns of HLAAs that each contain one or more PLAY HLAAs. In one case the entire sequence is viewed as programming and in

the other it is viewed as layout. It is quite possible that future analysis will reveal other situations where the reflection time should not be allocated to the category of the subsequent HLAA but rather should be viewed as reflecting upon what was just done, and thus allocated to the category of the preceeding HLAA. This is a question that needs to be pursued by researchers analyzing programmers' problem solving strategies.

4.9 HOW IMPORTANT IS AN HLAA'S ELAPSED TIME (I.E., THE TIME FROM THE START OF AN HLAA UNTIL THE LAST TIMESTAMP ON LOG ENTRIES STILL DESIGNATED AS PART OF THAT HLAA) AND HOW SHOULD IT BE TREATED?

Our analysis suggests that it is necessary to identify the conditions under which it is important. Each HLAA consists of one or more log entries. Each of these log entries has a timestamp and there is often a non-zero amount of elapsed time from the timestamp of the first of these entries until the timestamp of the last of these entries (unless of course the HLAA is represented by only one log entry as is the case for the save HLAA). We call this time the 'action time' for the HLAA. The action time for 96.4% of HLAAs is less than 100ms and 99.4% are less than 500ms. To help us decide if it is important where to allocate the action time of an HLAA (i.e., should we assume the programmer is thinking about what to do next or actively thinking about the action as it happens?) we computed the total action time and total reflection time for each of the categories of HLAAs. We found that the ratio of the total action time for programming HLAAs to the total reflection time allotted to programming HLAAs to be 0.0005, consequently, how we categorize the action time of programming HLAAs is of no consequence when trying to understand what programmers are doing when using Alice. Similarly, for scene layout HLAAs, excluding object animations using the right mouse button (see discussion below), the ratio is 0.0030, so even if we ignore this time or include it with layout time, the layout time will change by less about 0.3%.

For two specific HLAAs the action times are significant and thus deserved careful consideration. In the case of PLAY HLAAs, the total action time is 1.7 times the total reflection time preceding the PLAY HLAAs. For PLAY_ANIMATION HLAAs that are executed by using the right mouse button (object animations), the ratio is 0.40. Thus, failure to properly account for these action times could significantly alter conclusions about programmer behavior.

For PLAY_ANIMATION HLAAs, the action time is significant. The use of the right mouse button while in the scene layout editor of Alice (PLAY_ANIMATION HLAA) provides another way for programmers to position an object or object part (e.g., left hand

of robot or windup key of clown car), often giving more precision than using other parts of the Alice scene editor such as the mouse control buttons. In this mode, every object has a limited set of methods from which the object can respond; most objects have 'move', 'look at', and 'turn away from' in addition to other methods appropriate to that class of objects. These user actions result in usually short animations as the object executes the selected action and an often-lengthy sequence of log entries are written to the log file, but in some cases the animations can be many seconds. It seems safe to assume that in most cases the student watches the animation to see if the desired effect is achieved. Similarly, during the action time of a PLAY HLAA, students are probably watching the animation.

Thus, although the action time for programming HLAAs could safely be ignored, for consistency, when computing the total time for a category (layout, programming, and program play) we combine the sum of the action times and the reflection times associated with each HLAA in that category.

4.10 HOW CAN WE DETERMINE WHETHER GAP TIMES ARE DUE TO THE STUDENT REFLECTING OR TO THE STUDENT BEING OFF-TASK?

Baker (2007) found off-task behavior to be negatively correlated to learning, whereas reflection is an essential part of learning (Bransford et al, 2006). Thus, it is important to determine whether gap times are a time of reflection or inactivity. As a first step, we use a time-only single parameter model to determine a threshold above which the entire gap time period will be discarded. Baker developed a model for determining off-task behavior in a study of student close-ended mathematics problems using the Cognitive tutoring system (Baker, 2007). His model is composed of six parameters, the first of which was better at identifying off-task behaviors than a time-only single parameter model. He identified 'slow actions' (i.e., those actions with long reflection times) preceding or following 'fast actions' (i.e., those actions with short reflection times) as more indicative of off-task behavior than repeated 'slow actions.' In the future, we hope to investigate a more sophisticated model, than the single parameter model we used, for determining time-off-task. However, because the tasks (both open-ended and close-ended) our students performed are very different than those Baker studied (Baker, 2007), his six parameter model is not directly applicable.

We begin our investigation of off-task time by looking at the distribution of reflection times under the obviously false assumption that no time is "off-task" (i.e., using an

infinite cut-off threshold). The average reflection times and standard deviations for each category are shown in Table 6.

Table 6: Mean and standard deviation of reflection times in seconds assuming no time is "off-task".

HLAAs Category	mean	standard deviation
Layout	11	38
Programming	29	57
Program Play	21	77
Save	32	181

In our observations of our student population, 120 seconds was chosen to indicate a long enough time that students were probably "off-task." For programming and layout HLAAs, treating gaps that went for 120 seconds or longer as times during which the student is "off-task" would label a student as "on-task" during all gaps before programming and layout HLAAs within two standard deviations of the mean. For program play gap time, the threshold would need to be almost three minutes in order to retain all times within two standard deviations of the mean. "Reflection" time before a save is only allocated to the SAVE HLAA if this save is the last HLAA in the file. In all other cases, the time before a save is allocated to the category of the HLAA following the save on the assumption that most saves are in response to a periodic prompt from the system. The large standard deviation for SAVE HLAAs possibly reflects off-task time culminating in a save due to the prompt when quitting Alice.

To help us understand whether our initial key metrics were sensitive to the choice of what gap times should be considered off-task, we examined a range of upper thresholds (see Table 7). We see from the table that even using a large upper threshold (i.e., 10 minutes) has a noticeable impact on these key metrics.

Table 7: Key metrics for various upper thresholds with MLEG = 300ms.

upper threshold->	2min	3min	5 min	10 min	Infinite
prog time/layout time	1.613	1.676	1.703	1.699	1.627
avg prog reflect time (secs)	21.10	23.39	25.27	26.90	28.58
avg layout reflect time (secs)	7.89	8.53	9.12	9.76	10.84
total prog reflect (hours)	184.8	208.3	226.9	242.4	258.0
total layout reflect (hours)	114.1	123.8	132.8	142.3	158.2
total play reflect (hours)	43.2	51.1	58.7	66.7	89.0

Decisions about which upper threshold to use can be informed by observations of the students. For example, an analysis of video recordings we made of student pairs using Alice suggests that the threshold should be less than five minutes (O. Ruvalcaba, personal conversation, February 3, 2012). It is possible that pairs use Alice differently than solo programmers, but we do not have video recordings of solos programming with Alice to confirm this hypothesis. In summary, more work needs to be done in this area. Certainly video recordings would give us confidence about our decisions about time off-task.

4.11 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 were useful for understanding behavior in initial programming environments, we compared the key metric of the ratio of programming time to screen layout time for two different activities. As shown in Table 8, the metrics are very different for students using Alice when doing the Fairy assessment (FA) versus open-ended game programming. Since we expected students to do little, if any, layout when doing the FA, we were not surprised at the ratio of programming time to layout time for the FA and thus this comparison serves as a validation of our process.

Table 8: Comparing a key metric for Fairy Assessment (FA) vs. open-ended game programming.

	FA	Games	All
prog time/layout time	8.512	1.435	1.627

Table 9 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 (Kelleher, 2006). 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 combined the action times with the reflection times for these same HLAA categories. The fact that our values are somewhat similar to Kelleher's is an additional validation of our analysis process. The differences we see could be attributed to our allocation of the reflection time of SAVE HLAAs to the reflection time of the actions following the SAVE unless it is a SAVE prior to quitting Alice since Kelleher doesn't consider save low-level events in her calculations.

Table 9: Distribution of time by category from our analysis and that of Kelleher (2006).

	Ana2	Kelleher GA	Kelleher SA
Programming	39.0%	34.0%	48.3%
Layout	24.0%	40.8%	22.3%
Program play	37.0%	25.1%	29.3%
Other (not SAVE)	0.1%	0.1%	0.1%

5 DISCUSSION

This work has implications for tool developers implementing logging systems for programming environments and other software tools, as well as efforts to analyze logging data to understand how programmers are engaging with the environments. Here we make recommendations for logging systems. They are:

- Integrate the logging into the design of the software in a way that captures individual high-level user actions. The challenges involved in identifying the HLAAs for sequences of low-level log entries reduce the usefulness of the data for informing tool developers and educators. If the software was designed and built with high-level action logging where every user interface action was logged, then an entry would be logged for each high-level action and every high-level action could have a corresponding log entry or easily identifiable and unique sequence of log entries (e.g. beginAction … endAction).
- Include a 'start' log entry when the software opens. With Alice it is not possible to 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.
- 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.
- Consider the important questions to be asked from analysis of the logs before the design of the logging system is complete, and use them to dictate the fields to include in the logs. For example, we want to use the logs to determine whether novices use different successful problem solving strategies than experts. For example, researchers have used terms like tinkering and flailing to describe two different problem solving strategies. To see flailing in logs, the use of undo and redo must to be captured.

- Provide a mechanism to detect inactivity time. One approach is to set up the display to fade out after a certain period of keyboard or mouse inactivity. When a programmer makes a keystroke, a log entry is 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.
- Design and develop an open data repository for logs from novice programming environments as well as other learning 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 (Jadud, 2005; Jadud & Henriksen, 2009). Other programming environments designed for novices such as Scratch, Kodu, 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 (Koedinger et al, 2010). DataShop is an open data repository of logs from intelligent tutoring systems designed for the educational data mining community.
- The interface to the Alice programming environment has an undo/redo function. This provides an undo mechanism to the Alice programmer 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 programmer has done. Future logging systems should ensure that these important high-level user actions are captured in the log.

6 CONCLUSIONS AND FUTURE WORK

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

- We built a parser called Ana2 that can be used for the pre-processing of Alice log files. The output of Ana2 is a log of HLAAs where each HLAA corresponds to a function provided at the user interface level of the Alice programming environment instead of at the much lower level of operations found in the logging system built into Alice. This higher level representation facilitates pedagogical inquiry about the instructional worth of the Alice programming environment as well as investigation of learning strategies of the students using the environment. For example, researchers

can look for functionality provided by the Alice programming environment that is seldom used or that is seldom used successfully to identify parts of Alice that are difficult to use. If all initial programming environments had logging at the functional level, comparisons of various environments can be made that could support discussions about the 'best' programming environment for programmers of a specific age. Since Ana2 produces logs of HLAAs, the sequence of HLAAs in a log represents the sequence of operations by a student at the problem solving level. When the student was creating a computer game, this is a representation of the problem solving strategy used while working on an open-end task. Eventually, we hope this knowledge can be used to inform strategies for teaching and assessing the learning of novice Alice programmers as well as for improving the Alice programming environment.

- We described the process we used to create Ana2, a tool that takes as input the low-level Alice log files and translates them to high-level operations or HLAAs. Researchers can use this case study to learn about the fine-grained tuning that is necessary to make sense of low-level logging data. For researchers in the field of learning analytics, this kind of fine-grained tuning is necessary when usage logs are at a low level.
- We have demonstrated that Ana2 can distinguish between close-ended situations and open-ended ones. Our findings have the potential to provide insight into learner modeling for constructivist learning environments (Jonassen, 1999). In order to inform the development of tools to scaffold learning, it is important to know how novices think and the misconceptions they have. Computer log data can be used to describe what is actually happening in the process of interacting with the computer, and give insight into how educators and tool developers can make improvements. For example, learning analytics approaches can draw on cognitive learning theories that focus on how knowledge and skills are organized, rather than on whether a student has mastered a particular subject. To this end, these approaches can build on diSessa's (2006) suggestion to think of mistakes as a window into how students think, rather than as an indication of flawed knowledge.
- We made recommendations for the design of logging systems motivated by the problems we encountered with the pre-processing of the Alice logs. The improvements we suggest can be used to improve the Alice logging system and can be used when designing logging systems for other learning environments. Our recommendations should make learning analytics easier and more efficient.

- Logging data can contribute to the growing number of conversations about how to increase 'computational thinking' (Wing, 2010). It will allow us to address pressing research questions about what and how students learn from both open ended (e.g., exploratory activities) and closed-ended tasks. Relevant questions related to computational thinking build on unsupervised and supervised classification techniques similar to those described by Amershi et al. (Amershi & Conti, 2009):
    - Can 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?
    - Can logs be used to distinguish between problem solving strategies (e.g. flailing vs. tinkering) used by novices when working on close-ended assessments like the FA? Can a strategy such as flailing be identified in real-time? If so, we are on the path to developing a user-specific adaptive learning environment.
    - Can logs be used to distinguish between programming strategies used by students working in pairs versus those working alone?
    - Logs can show there are differences in the findings for the closed versus open-ended tasks. What do these differences mean?

In summary, we believe that what we have learned while pre-processing Alice logs can be applied to other domains of programming or problem solving as well as for creating and analyzing the logs generated by other software tools. The Alice HLAAs are relatively generic such that the use of learning analytics on HLAAs may discover learner strategies that are applicable to other novice programming environments. The transferability of learner strategies to other learning environments is an interesting question and would have to be researched. Additionally, there are implications for the development of logging systems in new or existing software. The lessons learned from the pre-processing of the Alice logs can be used to increase the transparency of logging data systems, thus increasing the potential accuracy and depth of the findings and decreasing the amount of time needed to parse the data and do the analyses. Finally, our method of pre-processing of Alice logs can inform others of how to prepare logging data for analysis—the required steps and decisions that need to be made before analyses can begin.

ACKNOWLEDGEMENTS

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. Ana 2.0 is based on the initial version of Ana created by Ben Smith as part of his computer science senior thesis research at UCSC. We thank him for his continued conversations with us about Alice logging. We also thank Shannon Campe and Eloy Ortiz for their important role in running the classes and collecting the data. We thank all of the teachers, teaching assistants, and students involved.

REFERENCES

ALEXANDER, P. 2003. The Development of Expertise: The Journey from Acclimation to Proficiency. *Educational Researcher*, *32*, 10-14.

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.

BAKER, R.S.J.D. 2007. Modeling and Understanding Students' Off-task Behavior in Intelligent Tutoring Systems. *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '07).* New York, NY: ACM, 1059-1068.

BAKER, R.S.J.D., and 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.

BRANSFORD, J.D., BARRON, B., PEA, R.D., MELTZOFF, A., KUHL, P., BELL, P., STEVENS, R., SCHWARTZ, D.L., VYE, N., REEVES, B., ROSCHELLE, J., and SABELLI, N. 2006. Foundations and Opportunities for an Interdisciplinary Science of Learning. In R.K. Sawyer (Ed.), *The Cambridge Handbook of the Learning Sciences*. 19-34. New York: Cambridge University Press.

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.

BURNETT, M., COOK, C., PENDSE, O., ROTHERMEL, G., SUMMET, J., and WALLACE, C. 2003. End-user Software Engineering with Assertions in the Spreadsheet Paradigm. *In Proceedings of the 25th International Conference on Software Engineering,* 93-103. IEEE Computer Society.

CHOQUET, C., and IKSAL, S. 2007. Modeling Tracks for the Model Driven Re-engineering of a TEL System. *Journal of Interactive Learning Research,* 18(2), 161-184.

CONWAY, M.J. 1997. Alice: Easy-to-Learn 3D Scripting for Novices (Doctoral dissertation, University of Virginia).

DANN, W.P., COOPER, S.P., and ERICSON, B. 2009. *Exploring Wonderland: Java Programming Using Alice and Media Computation (1st ed.).* Upper Saddle River, NJ: Prentice Hall Press.

DIMITRACOPOULOU, A. 2005. Designing Collaborative Learning Systems: Current Trends & Future Research Agenda. In *Proceedings of the 2005 Conference on Computer Support for Collaborative Learning: Learning 2005: The next 10 years!* 115-124. International Society of the Learning Sciences.

DISESSA, A.A. 2006. A History of Conceptual Change Research: Threads and Fault Lines. In R.K. SAWYER (Ed.), *The Cambridge Handbook of the Learning Sciences,* 265-281. New York: Cambridge University Press.

FINCHER, S., and UTTING, I. 2010. Machines for Thinking. *Transactions on Computing Education,* 10(4), Article 13, 1-7.
HATANO, G. 1988. Social and motivational bases for mathematical understanding. In G.B. SAXE and M. GEARHART (Eds.), *Children's Mathematics,* 55–70. San Francisco, CA: Jossey-Bass.
JADUD, M.C. 2005. A 1st Look at Novice Compilation Behavior. *Computer Science Education,* 15(1), 25-40.
JADUD, M.C., and HENRIKSEN, P. 2009. Flexible, Reusable Tools for Studying Novice Programmers. In *Proceedings of the Fifth International Computing Education Research Workshop (ICER '09).* New York, NY: ACM, 37-42.
JONASSEN, D. H. 1999. Designing Constructivist Learning Environments. *Instructional Design Theories and Models: A New Paradigm of Instructional Theory,* 2, 215-239.
KELLEHER, C. 2006. Motivating Programming: Using Storytelling to Make Computer Programming Attractive to More Middle School Girls (Doctoral dissertation). Retrieved from reference http://www.dtic.mil/dtic/ number ADA492489.
KOEDINGER, K. R., BAKER, R.S.J.D., CUNNINGHAM, K., SKOGSHOLM, A., LEBER, B., and STAMPER, J. 2010. A Data Repository for the EDM Community: The PSLC DataShop. in C. ROMERO, S. VENTURA, M. PECHENIZKIY, R.S.J.D. BAKER (Eds.) *Handbook of Educational Data Mining*, 46-53. Boca Raton, FL: CRC Press.
MACFADYEN, L.P., and DAWSON, S. 2010. Mining LMS Data to Develop an "Early Warning System" for Educators: A Proof of Concept. *Computers & Education*, 54, 588-599.
MCDOWELL, C., WERNER, L., BULLOCK, H.E, and FERNALD, J. 2006. Pair Programming Improves Student Retention, Confidence, and Program Quality. *Communications of the ACM,* 49(8), 90-95.
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).* New York: NY, 39-46.
PIECH, C., COOPER, S., SAHAMI, M., KOLLER, D., and BLIKSTEIN, P. 2012. Modeling How Students Learn to Program. In *Proceedings of the 43st ACM Technical Symposium on Computer Science Education (SIGCSE).* ACM, New York, NY, USA.
ROBINS, A., ROUNTREE, J., and ROUNTREE, N. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137-172.
SMITH, B. 2010, June. Effect of Pair Programming on Middle Schoolers Using Storytelling Alice. Poster session presented at the *13th Annual UCSC Undergraduate Research Symposium*, Santa Cruz, CA, USA.
SOLOWAY, E., EHRLICH, K., BONAR, J., and GREENSPAN, J. 1983. What do Novices Know about Programming? In B. SHNEIDERMAN and A. BADRE (Eds.), *Directions in Human–Computer Interactions,* 27-54 . Norwood, NJ: Ablex.
SOLOWAY, E., and SPOHRER, J. C. (Eds.). 1989. *Studying the Novice Programmer.* Hillsdale N J: Lawrence Erlbaum.
UTTING, I., COOPER, S., KÖLLING, M., MALONEY, J., and RESNICK, M. 2010. Alice, Greenfoot, and Scratch -- A Discussion. *Transactions on Computing Education*, 10(4), 1-11.
WERNER, L., DENNER, J., CAMPE, S., and KAWAMOTO, D.C. 2012. The Fairy Performance Assessment: Measuring Computational Thinking in Middle School. *Proceedings of the 43st ACM Technical Symposium on Computer Science Education (SIGCSE).* ACM, New York, NY, USA.
WING, J.M. 2011, Spring. Computational Thinking: What and Why? In *Link Magazine*, 6.0. Retrieved from http://link.cs.cmu.edu/article.php?a=600.