Open-Ended Novice Programming Behaviors and their Implications for Supporting Learning

Michelle Ichinco
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA, USA
michelle_ichinco@uml.edu

Caitlin Kelleher
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO, USA
ckelleher@wustl.edu

***Abstract*—Though support for learning computing in schools is growing, many children still begin learning to program without formal support in open-ended programming environments. While researchers have evaluated the final code of these types of projects, we know little about how users' behaviors and usage of support tools relate to understanding. We ran a study where participants had open-ended programming time with access to one of two support tools: suggestions or tutorials. Participants then completed four tasks which required understanding of the suggestion or tutorial content. We did not find an effect of suggestions compared to tutorials on knowledge application, but we did find that many participants who performed better tended to explore more of the interface, code behaviors, and support tools. Our results suggest that future tools for encouraging learning during open-ended programming should likely focus on supporting users who tend to explore less on their own.**

***Index Terms*—Novice programming, support tools**

I. Introduction

Introduction to programming often still takes place with minimal or no formal instruction in one of the many constructionist novice programming environments. Outside of classrooms, tools like Scratch [23], App Inventor [21], Snap [27], Alice [5], and Looking Glass [8] fundamentally rely on the interest, motivation, and exploration of their users. These systems have demonstrated immense success in terms of engagement [16]. Yet most of the research attention for supporting young novice programmers has focused on developing concrete pedagogical activities, rather than supporting learning when users have control of their own activities. Our work aims to understand what kinds of open-ended coding behaviors predict understanding or learning and the potential implications for designing tools that support independent learning. Specifically, this paper answers the previously unaddressed question of how novices' interactions with support tools, the programming environment, and code blocks during open-ended programming relate to their abilities to demonstrate knowledge application through tasks.

To date, research on open-ended programming has focused on analysis of users' final shared projects, rather than their behaviors. These evaluations can help researchers and educators gauge learning and also help learners understand their own progress [18]. Users' final code can begin to indicate learning, but cannot take into account the whole context of the user's behaviors or their ability to use code blocks effectively on specific tasks [17], [25], [32]. In order to design tools to support novices in open-ended programming, we observed novices' coding behaviors, interactions with the programming environment, and usages of two existing support tools.

To understand how novices interact with support tools during open-ended programming, participants could access one of two support tools: suggestions or tutorials. These options represent two of the most common and effective existing support tools for novice open-ended programming. Many novice programming environments provide tutorials. We designed our tutorials based on Scratch 2.0, which provided short videos demonstrating the steps to create a simple animation. Recent studies have also shown that suggested code examples can be highly effective in encouraging novices to explore and add new code blocks during open-ended programming [11]–[13]. However, no research has yet explored how users' interactions with these support tools relates to their behaviors and abilities to apply their knowledge on tasks.

We ran a study in which young novice programmers had time for self-directed open-ended programming followed by knowledge application tasks. During part of this open-ended programming time, participants either had access to suggestions or tutorials that demonstrated use of two code blocks: simple parallel execution *Do together* or the basic loop *Repeat loop*. Following the open-ended programming time, participants worked on four knowledge acquisition tasks. In order to correctly complete the tasks, participants had to correctly apply *Do together* and/or *Repeat loop* code blocks to modify animations. These two code blocks are most commonly used by early novice programmers and are most likely to be accessible in the relatively short span of a lab study.

We believe that this is the first work to look at the connection between young novice programmers' open-ended programming behaviors and their abilities to apply knowledge to complete tasks. Our results indicate that those who applied constructs more correctly on tasks likely explored the available code blocks more often and interacted more often and deeply with code constructs and support tools. Supporting novices who explore less on their own may encourage more learning in open-ended novice programming environments.

II. Related Work

Recently, researchers have begun to evaluate and build tools to support open-ended programming. A variety of tools support and analyze novice programming through specific tasks. This work is most related to research surrounding open-ended programming. We build upon the work in: 1) modeling computational skill, 2) analysis of code and behavior, and 3) supporting open-ended novice programming.

A. Models of computational skill

Some methods of evaluating novice code projects are rooted in theories of computational thinking. The assessment of computational thinking is still a highly active research area, as described by Basso et al., though primarily in classroom settings [2]. Evaluations of open-ended programming have recently begun to leverage computational thinking concepts and frameworks [18]. Because analyses of code projects typically take place at a large scale and long after the creators have completed the projects, they focus on Brennan and Resnick's computational thinking *concepts* and cannot take into account practices and perspectives [4]. Our work begins to look at novices' behaviors, which more closely align with *practices* and *meta-skills* discussed in computational thinking literature. These elements of computational thinking have yet to be explored outside of a classroom context.

B. Analysis of code and behavior

Several studies have looked at the overall skill progression of programmers as a group by evaluating novices' final projects. Some have looked primarily at the numbers and types of blocks in users' final projects and have found an upward trend in the number of types of code blocks programmers use over time [17], [25], [32]. Researchers have also found that many programs are relatively short, rarely use abstractions, and suffer from code smells [1], [10]. Another study of the specifics of open-ended code looked at naming of variables and procedures, finding that they may be problematic [28]. Issues in blocks-based code like code smells can have a real negative impact, especially when they are shared to online communities, where other novices may try to reuse code [30].

Very little work has investigated the behaviors of novices during open-ended programming. Researchers have begun to profile general interaction styles and found high-level differences in programming approaches. Groups of users sometimes focus on visual, story, or animation elements in Alice [6]. Some work has looked at novice behaviors during *open-ended assignments*, such as exploring the ways students tinker [7]. This differs from open-ended programming because assignments still provide a goal that the student needs to accomplish. We expect novices' behaviors and needs for support for learning to be very different when the novice has complete control over their goals.

Instead of focusing on novices' final code projects, we analyze their *specific behaviors*, regarding the code, the interface, and the support tools, and how those behaviors relate to novices' abilities to apply their knowledge in certain tasks.

C. Supporting open-ended novice programming

Open-ended programming environments typically provide static support through a set of available tutorials or documentation and recent systems aim to actively support open-ended programming. Dr. Scratch, Quality Hound, and CodeMaster enable novices to evaluate their own programs to determine their computational thinking skills or identify potentially problematic code smells based on rubrics [18], [19], [29], [31]. The Example Guru also analyzes novice program code incrementally during the programming process and provides a list of suggested code snippets, which novices access more than static support [11]–[13]. Researchers have also developed a wide variety of ways of supporting novice programmers working on specific and open-ended tasks [14], [15], [22], but they rely on knowledge of the tasks in order to nudge the novice toward a solution. None of these tools takes into account users' behavior and interactions.

As part of the push for CS education in schools, much of the work on evaluating and supporting novice programmers has focused around curricular and task-based contexts. So far, evaluation of open-ended novice programming has focused on novices' final code projects. We are unaware of research that looks at how novices' coding behaviors and interactions with support relate to their skill or how those behaviors impact open-ended novice programmers' needs for support.

III. System

To capture open-ended programming, we chose to have participants use Looking Glass because many children do not have exposure to it, compared to well known systems like Scratch. Due to this, we expect novices' behavior during the study may contribute more to their end performance. Throughout their open-ended programming we wanted them to have the best available kinds of support: tutorials and suggestions. Most systems today use tutorials to provide support. A recent study suggests that code-based suggestions may be more effective [13]. We describe Looking Glass and both forms of support.

A. Looking Glass Programming Environment

Looking Glass is a blocks programming environment designed for children to make 3D animations, based on Storytelling Alice [16]. It contains many similar functionalities as other blocks programming environments for novice programmers. Looking Glass users code by dragging and dropping blocks (see Fig. 1). Looking Glass has a unique animation API that includes methods such as *move*, *turn*, or *resize* and also contains code concepts like a simple parallel execution block *Do together* and a simple *Repeat loop*. Looking Glass has a built-in documentation feature accessible from a 'more' button on each code action block added to a program (not on *Do together* or *Repeat loops*).

B. Code-Based Suggestions

Suggestions provide ideas for ways a user could integrate a certain code block into their program. They provide two examples of a specific animation using that code block. We

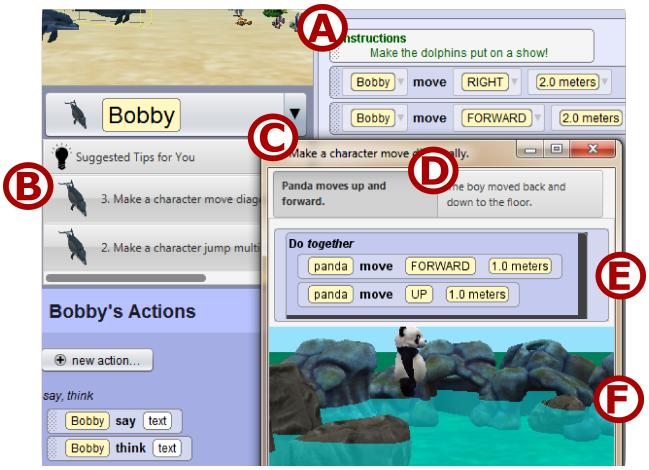

Fig. 1. (A) Looking Glass. (B) Suggestion list. (C) Accessed suggestion. (D) Two code examples. (E) Code snippet, which users can hover over to learn more. (F) Example code execution video.

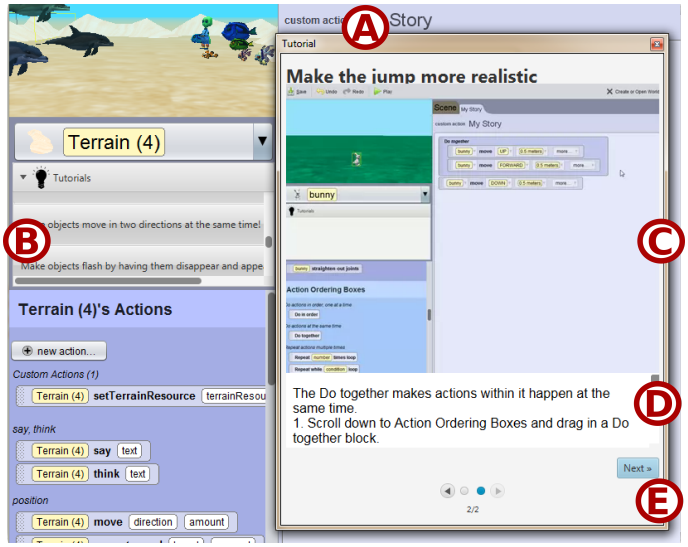

Fig. 2. (A) Accessed tutorial. (B) List of tutorials, which always has the same set of 13 tutorials. (C) Short video that shows how to complete the step. (D) Text instructions. (E) Button to go to the next step.

describe the interface, the rules that trigger the suggestions, and the set of suggestions we used.

1) Suggestion interface: Users can access suggestions from a list near the code block menu (see Fig. 1-B). Fig. 1-C shows a suggestion a user has accessed for 'Making a character move diagonally'. It has two examples: 1) a panda moving forward and up at the same time, and 2) a boy moving backwards and down at the same time. Both suggestions have a title and a snippet of example code. Mousing over the code block shows a description of how the code works and where to find the code block in the code menu. Users can execute the code and view the output below the code (see Fig. 1-F).

2) Suggestion rules: When a user executes their code, the system runs 'rule' scripts, which check the code for conditions that would trigger suggestions. Each suggestion has a rule for when it should trigger. For example, for the suggestion 'Make a character move diagonally', the rule checks a user's code for two 'move' code blocks in sequence which have the same object (like panda) and have two different directions (up and forward, for example), but no *Do together* code block.

3) Suggestion and rule set: The set of suggestions and rules in this study were generated based on a repository of programs and a semi-automatic method from existing research [13]. This process resulted in 7 suggestions for the *Repeat loop* and 73 suggestions for the *parallel execution* blocks.

C. Static Tutorial Set

The static set of 13 tutorials provided content demonstrating how to use the *Do together* and *Repeat loop* code blocks. Fig. 2-B shows the list where users could access the tutorials, which is visually almost identical to the list of suggestions. When a user accessed a tutorial from that list, they could view a short video for each step in creating a short animation and short text explanations, as shown in Fig. 2-A. We based the design of our static set of tutorials on Scratch [26]. We created the content for the tutorials based on a set of previously successful hand-authored suggestions [11].

IV. Evaluation

We ran a study in which participants had time for open-ended programming and then completed knowledge application tasks. We use this study to answer three questions: 1) did interactions with or availability of suggestions relate to knowledge application, 2) did behaviors during open-ended programming relate to knowledge application, and 3) did code use and interaction during open-ended programming relate to knowledge application?

A. Study Protocol

The study had five phases: 1) default open-ended, 2) support intro tasks, 3) supported open-ended, 4) knowledge application tasks, and 5) surveys.

1) Default Open-Ended: The fifteen minute default open-ended phase enabled us to gather data on the types of code participants used before having access to tutorials or suggestions. Participants could create any animation they wanted using one or more of nine pre-made scenes. We used pre-made scenes, designed through pilot studies, because in the scope of a short lab study, we wanted participants to focus on coding rather than creating the scenes. If a participant requested help with the mechanics of blocks coding, a researcher demonstrated how to drag and drop code blocks and execute code.

2) Support Intro Tasks: The two five-minute support intro tasks aimed to ensure that participants were aware of suggestions or tutorials, depending on the participant's condition. Each task asked the participant to modify an API method call. The corresponding suggestion or tutorial demonstrated how to complete the task. If a participant got stuck on a support intro task, a researcher guided them to complete the task using the available help. If a participant completed a task without the suggestion or tutorial, the researcher demonstrated how they could have accessed the support tool to solve the task.

3) Supported Open-Ended: The goal of the thirty minute supported open-ended phase was to understand how participants would choose to access and interact with suggestions or tutorials. Participants had access to suggestions or tutorials, depending on their condition. Suggestion participants received a set of suggestions based on their code. Tutorial participants had access to 13 tutorials. We told participants that they were not required to access or use them. We first asked participants to create a performance animation using a specific template with dolphins underwater. We gave participants this prompt because in pilot studies, some participants had a hard time generating ideas. They then could choose from the same set of pre-existing templates as the default open-ended phase to create their own animations. Participants could choose when they had completed any animation and move on to another animation, return to a previous animation, or end the phase.

4) Knowledge Application Tasks: Participants completed four three-minute tasks that evaluated their understanding of the *Do together* and *Repeat loop* code blocks. The four tasks asked participants to create specific animations but did not provide instructions. To successfully complete the tasks, participants had to know where to place the following code blocks for each of the four tasks: 1) *Do together*, 2) *Repeat loop*, 3) *Do together* nested within *Repeat loop*, and 4) *Repeat loop* nested within *Do together*. Participants received these tasks in an order balanced across participants. Based on our pilot study, we minorly adapted these tasks from a study evaluating these concepts [9].

5) Surveys: Participants filled out a survey about their age, gender, and programming experience. Participants filled out a survey at the end of the study about their experience coding and the usefulness and understandability of suggestions or tutorials.

B. Participants

We recruited 62 participants from a local science-focused mailing list. We had a pilot study with 18 participants to resolve issues with the materials and methods. The remaining 44 participants completed the final study. We exclude four participants: three because they had participated in similar studies with our lab and one due to technical issues. This paper presents the results from the remaining 40 participants aged 8 to 15 ($M = 11.2, SD = 1.6$), of whom 24 were male and 16 were female. Gender, age, and programming experience have, in existing work, affected early learning in coding [9]. Although we attempted to recruit participants with no or minimal programming experience, upon arrival, we discovered that many children had more. In order to attempt to get fair results across conditions, we did our best to balance the gender, age, and self-reported programming experience across conditions. Both conditions had average ages of 11.2 and 11/20 participants with minimal (less than 3 hours) or no programming experience. The suggestion condition had 9/20 females, and the tutorial condition had 7/20 females.

V. Data and Analysis

To understand how participants' behaviors during open-ended programming corresponded with their ability to apply knowledge within tasks, we analyze: knowledge application, open-ended programming behaviors, and demographics. We do not analyze the support intro tasks because they were informational for the participants, rather than for evaluation.

A. Knowledge Application

Our tasks aim to evaluate participants' ability to apply their knowledge of *Do together* and *Repeat loops*. We scored tasks using rubrics for each task and also broke users into non-, low-, mid-, and high-performers for more in-depth analyses.

1) Scoring rubric: We designed rubrics to score participants' attempts for the four tasks. For each task, participants received points for adding correct *Do together* and *Repeat loop* code blocks, moving the correct code blocks into the *Do together* and *Repeat loop* code blocks, and not having extra code blocks within the constructs or added to the program as a whole. These scores can be calculated objectively, as they are merely a count of the correct number and placement of code blocks. The tasks scores are represented as percentages of the total number of available points for that task.

2) Performance groups: To explore how coding behavior and suggestion and tutorial interactions relate to knowledge acquisition, we split users into four groups based on performance on tasks: **non-performers**, **low-performers**, **mid-performers**, and **high-performers**. We did this because we noticed there were distinct ways participants performed: the twelve non-performers added no correct code blocks to any of the four tasks; the eleven low-performers had at least one correct code block added to at least one task, but none fully correct; the eight mid-performers all had exactly one task correct, which involved only the *Do together* code block; and the nine high-performers had at least two tasks fully correct.

B. Open-ended programming behaviors

We analyze both open-ended phases together for this paper to provide more data about how participants interacted with the interface and code in addition to the support tools. We collected and analyzed two types of data: code interactions and interactions with the programming environment.

1) Code interactions: We recorded the *Do togethers* and *Repeat loops* participants used, as the suggestions and tutorials introduced these and they are the concepts the tasks test. For these blocks, we collected: 1) how many each participant used during open-ended programming, 2) how many times they made code changes within the blocks, and 3) how often users nested these code blocks within each other or within other code blocks.

2) Interface Actions: We collected the segments of users' actions during open-ended programming and the total time each user spent performing each type of action. Table I lists and describes the actions we collected. We recorded the time as the time the user started doing an action of that type until the time they did an action of a new type. If the time recorded was less than a second, we logged it as one second in order to account for the user having taken the action.

3) Support actions: We wanted to explore whether different ways of interacting with support tools may have had a relationship with knowledge application. We captured the specific ways users interacted with suggestions and tutorials. This included whether users scrolled through the lists of suggestions or tutorials, when they opened them, and when they closed them. For suggestions, we also recorded whether they executed the example code (tutorial examples executed automatically), and whether they switched between the two examples. For the tutorial, we measured whether users clicked to switch to other steps within the tutorial.

C. Demographics

We captured participants' age, gender, and programming experience. Each of these features has the potential to impact task success. Age during middle school can have an impact on task performance due to critical changes that occur in abstract processing [20]. Researchers have found differences in how males and females complete computing tasks [3]. Different types and lengths of programming experience could affect whether participants already knew programming concepts before starting the tasks.

D. Threats to validity

Our primary threat to validity is our population. Our recruitment through a STEM-focused mailing list may mean participants had more access to STEM resources than the average 8-15 year old. Participants may have answered questions about their past coding experience incorrectly, either purposefully or not. Though we tried to balance age across conditions, our study included children before and after critical developmental stages [20], which may have affected our results. Our results may also not generalize to adult non-programmers or programming environments with different types of support tools.

VI. Results

To better support novices in open-ended programming, we explored how interactions with support tools, the programming environment, and specific constructs related to knowledge application. Our results answer three questions: Q1) Did interactions with or availability of suggestions or tutorials relate to knowledge application, Q2) Did coding behaviors during open-ended programming relate to knowledge application, and Q3) Did code used during open-ended programming relate to knowledge application?

A. Q1: Did interactions with or availability of suggestions or tutorials relate to knowledge application?

We first analyzed whether having access to suggestions or tutorials had an effect on task success. In order to design support for open-ended programming, we explored more deeply novices' interactions with suggestions and tutorials and how this related to their success on tasks.

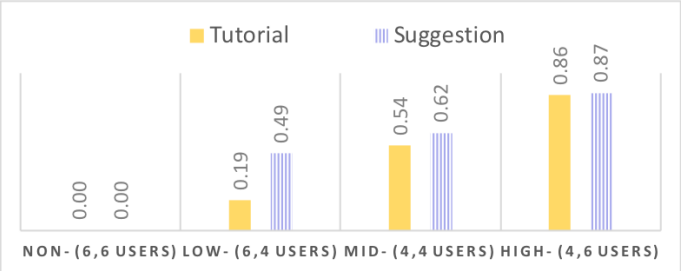

Fig. 3. Average knowledge application scores across groups

1) Did availability of suggestions or tutorials that introduced code concepts affect performance on tasks?: We did not find a significant difference between participants who had access to suggestions or tutorials and their abilities to complete tasks. We used a MANCOVA with Roy's largest root to evaluate the effect of condition and task on task scores. We used age, gender, and whether participants had programming experience as covariates. Participants did not significantly differ on task scores with condition ($Roys\ largest\ root = .12, F(1,33) = 0.9, p > .1$). For our demographic covariates, we only found an effect of age on task scores ($Roys\ largest\ root = .40, F(1,33) = 2.97, p < 0.05$). We followed up with a correlation to determine how strong of a relationship age had with knowledge application. We found age to have a moderate positive correlation with knowledge application ($r = 0.38, p < 0.05$). This aligns with prior findings on age and success on programming tasks.

We wondered whether there were differences in access to or preferences about suggestions or tutorials that may help to better explain this result. Though the number of suggestions varied across participants since they were context sensitive, we do not believe this affected the results. Suggestion participants received on average 9 suggestions during open-ended programming ($SD = 4.5, range = (1, 17)$). Tutorial participants had access to 13 tutorials. Participants rarely accessed more than a few suggestions or tutorials. Participants also did not significantly differ in their post-study survey responses about the study or suggestions and tutorials. There was no significant difference between suggestion and tutorial user responses about how they perceived programming in the study or whether suggestions were useful or understandable.

We also looked at task scores within each of the non-, low-, mid-, and high-performing groups, as shown in Fig. 3. After splitting into four groups, our sample sizes are too small to determine statistical differences, but average scores across groups may indicate future directions to explore. Notably, the tutorial low-performers had an average score of 0.19 ($SD = .1$) across all tasks and participants. Suggestion low-performers had an average score of 0.49 ($SD = .2$). This means suggestion low-performers were closer to completing tasks on average. Interestingly, much of the suggestion interaction was concentrated in the low-performing group. Tutorial interactions are mainly consistent across groups, as shown in Fig. 4.

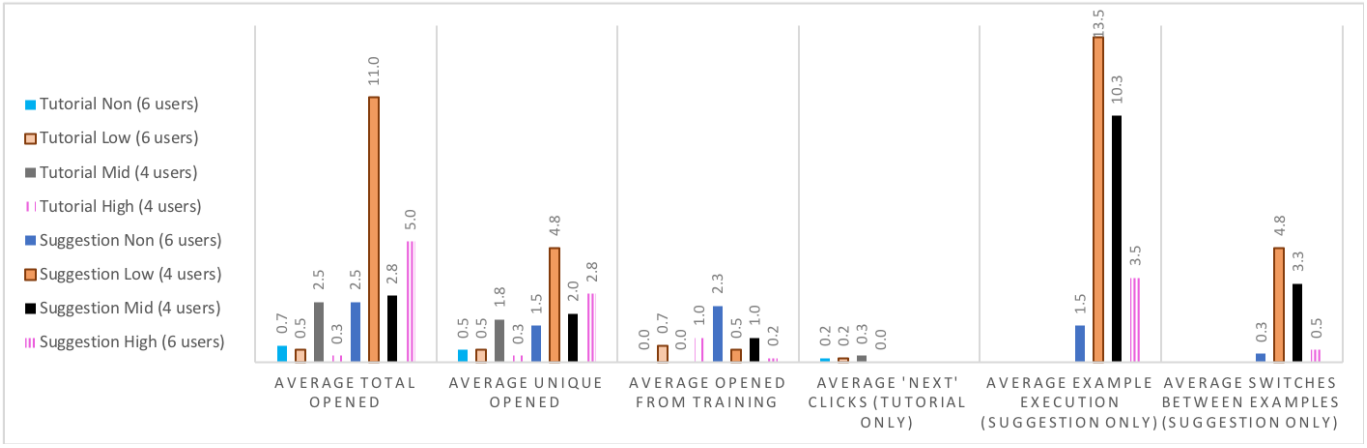

Fig. 4. Suggestion and tutorial interactions by knowledge application group. Suggestion users interacted much more with the support tools, especially in the low- and mid-performer groups.

2) Interactions with suggestions and tutorials: We explored how participants interacted with suggestions and tutorials by knowledge application group (non-, low-, mid-, and high-). Each of the groups had close to even splits between the number of tutorial and suggestion users (see Fig. 3). Suggestions specifically may have helped low- and mid-performers. Non- and high-performers seemed to interact with them less.

Low-performers and mid performers interacted with suggestions much more than non- or high-performers. The four low-performers who had access to suggestions opened on average 11 suggestions, executed examples on average 13.5 times and switched between the primary and secondary examples on average 4.8 times, as shown in Fig 4. Mid-performers only opened on average 2.8 suggestions, but they executed examples on average 10.3 times and switched between primary and secondary examples on average 3.3 times. Correspondingly, low- and mid-performers rated the usefulness of suggestions as 6.75 and 5 and the understandability of suggestions as 6.75 and 6 out of 7, respectively. In contrast, non- and high-performers had fewer interactions with suggestions after accessing them. It's possible that non- and low-performers began the study with similar knowledge and those who chose to interact more with suggestions then could begin to be able to apply knowledge of *Do togethers* and *Repeat loops* to tasks. The high-performers in each group likely already had some experience with the concepts shown in the suggestions and tutorials, so they may not have felt the need to view or interact with more of them. Corresponding with the lower interactions, non- and high-performers rated the usefulness of suggestions as 4 and 4.4, and the understandability of suggestions as 4.7 and 5.2, respectively.

Tutorial participants did not display this trend as much. Mid-performers opened on average 2.5 tutorials, compared to 0.7, 0.5 and 0.25 for non, low, and high-performers respectively. Few participants in any group clicked the 'next' button to get to a second or third step of a tutorial. The second step of each tutorial demonstrated adding the *Do together* or *Repeat loop* blocks, so almost none of the participants actually viewed this within tutorials. Tutorial post-study survey responses did not show a strong relationship with interaction: non-, low-, mid- and high-performers rated the usefulness of tutorials as 6.8, 5.2, 4, and 5.5 and the understandability of tutorials as 5.8, 4.5, 6, and 6 out of 7, respectively.

Participants' previous programming experience may have been a major determining factor within the short span of our study. Participants had only a half an hour of open-ended programming time with access to suggestions or tutorials. Those who saw a *Do together* or a *Repeat loop* for the first time in our study were unlikely to master them in such a short period of time. The huge amount of interaction low-performers had with suggestions and their higher scores than tutorial low-performers may indicate that suggestions benefited those with the least programming experience who were most willing to explore. Because low-performers did not complete any of the tasks, they likely did not already know how to use *Do together* or *Repeat loop* code blocks. Low-performers' choice to interact with suggestions may be what ended up distinguishing them from the non-performers.

B. Q2: Did behaviors during open-ended programming relate to knowledge application?

We analyzed the relationship between time spent performing programming environment actions and knowledge application scores using Pearson's correlation coefficient. We also investigated whether exploring actions related to support tool use.

Participants' exploring actions correlated with successful knowledge application, while program execution negatively correlated with successful knowledge application, as shown in Table I. We found a moderate positive correlation between the time spent exploring the available code blocks menu and task success ($r = 0.43, p < 0.01$). Use of the 'more' button had a marginally significant and somewhat weaker correlation with task success ($r = 0.3, p = 0.06$). The 'more' button allows users to add arguments to their code like the speed of an action. In contrast, the time participants spent executing their code had a negative correlation with knowledge application

TABLE I
ACTIONS AND CORRELATION WITH KNOWLEDGE APPLICATION

Action Type	Description	Cor. w/ knowledge application
Explored Support	Scrolled suggestion or tutorial list	$r = 0.18$
Support	Interacted w/ suggestions or tutorials	$r = 0.02$
Executed	Executed their program	$r = -0.39*$
Explored Code Blocks	Scrolled through menu of available code blocks	$r = 0.43**$
More	Clicked 'more' button on a code block	$r = 0.3, p = 0.06$
Documentation	Interacted with in-application documentation	$r = -0.10$
Inserting	Inserted a code block	$r = 0.11$
Deleting	Deleted a code block	$r = -0.11$
Undo/Redo	Undid or redid a code change	$r = -0.21$

$*p < 0.05$ $**p < 0.01$

TABLE II
SUGGESTION INTERACTION RELATIONSHIP WITH INTERFACE ACTIONS

Interaction with suggestion	Correlation w/ exploring	Correlation w/ execution
# suggestion example executions	$r = 0.69**$	$r = -0.35$
# switches between suggestion examples	$r = 0.49*$	$r = -0.33$
# Suggestions opened	$r = .31$	$r = -0.23$
# Unique Suggestions opened	$r = .29$	$r = -0.18$
Times went to next step in tutorial	$r = 0.48*$	$r = -0.37$
# Tutorials opened	$r = 0.1$	$r = -0.26$
# Unique tutorials opened	$r = 0.06$	$r = -0.32$

$*p < 0.05$ $**p < 0.01$

($r = -0.39, p < 0.05$). Those who explored more may be 'tinkerers', which researchers suspect may have educational benefits and may lead to better debugging performance [3], [24]. The negative correlation between time spent executing code and knowledge application was surprising, as code execution might also seem to correspond with tinkering behaviors. Executing code may be related to an ineffective type of tinkering which distracts from more substantive behaviors [3].

Due to the correlations between exploring and executing code with task success, we wondered whether these behaviors related to interaction with support tools. Suggestions have increased exploration of new code [11], [12]. The number of suggestions or tutorials participants opened did not relate to exploration of the programming environment, but interactions with them did (see Table II). Executing examples had a strong correlation with exploring ($r = 0.69, p < 0.01$). Switching between examples within a suggestion also had a moderate correlation with exploring ($r = 0.49, p < 0.05$). Going to the next step within a tutorial had a moderate correlation with exploration ($r = 0.48, p < 0.05$). These correlations may suggest that rather than current support **encouraging exploration**, they support participants who **naturally explore**. Both suggestions and tutorials themselves require some amount of exploration and interaction in order for users to fully benefit from them.

Total time spent on open-ended programming may relate to task success. We found a marginally significant correlation between total time spent programming and average knowledge application ($r = 0.31, p = 0.06$). Due to a technical error, two participants spent extra time in the baseline phase (16.6 and 19.5 minutes, compared to the typical 15 minutes) and one participant spent 44 minutes in the supported open-ended phase (typically 30 minutes). We allowed participants to end their open-ended time early if they did not want to keep working on their animations. Several chose to end the 30 minute supported open-ended phase slightly early and one participant spent 16 minutes. On average, participants spent 29.75 minutes in the supported open-ended phase ($SD = 3.3$).

C. Q3: Did code used during open-ended programming relate to knowledge application?

As interactions with code blocks may be one form of exploration of how to use them, we wanted to explore whether users' interactions throughout open-ended programming related to their knowledge application. We analyzed correlation of knowledge application with: 1) final usage of code in programs, and 2) interactions with code.

1) Do together and Repeat usage: We found a correlation between the number of *Do together* and *Repeat loop* code blocks in users' final programs and their knowledge application. The correlation of the sum of total *Do together* and *Repeat loop* constructs in participants' final projects had a moderate significant correlation with their average task score ($r = .55, p < .001$). This further supports existing literature that has found that usage of code within a users' program indicates understanding or learning [19].

2) Interaction with Do together and Repeat loop blocks: Beyond the insertion of specific code blocks, we wanted to better understand how participants interacted with the code blocks. We analyzed several behaviors: inserting or removing code from the code blocks, deleting the code blocks, or nesting combinations of the code blocks.

We measured the number of times participants inserted a code block into a *Do together* or *Repeat*, how many times they moved a code block into or out of one of these blocks, and how many times they deleted a code block directly from a *Do together* or *Repeat* block. We summed these together as one metric of 'interactions with the code constructs' and found a strong correlation between the total number of interactions users had with *Do together* and *Repeat* blocks and average task score ($r = .61, p < 0.001$). Table III-Interactions shows the average numbers of interactions participants had with the code blocks across the four groups.

Participants did not delete or nest many *Do together* or *Repeat loop* code blocks, so we did not analyze whether the numbers of these blocks corresponded with knowledge application. However, based on groups, those with higher knowledge application scores were more likely to have deleted at least one of these code blocks, as shown in Table III. We also saw a general increase in participants who nested *Do together* or *Repeat loop* code blocks as participants performed better on knowledge application tasks. Interestingly, the trends in these two behaviors were very similar, as shown in the

TABLE III
INTERACTIONS WITH CONSTRUCTS

Group	Total Do togethers	Total Repeat loops	Interactions	# who deleted	# who nested
Non-performers	M=.08, SD=.3	0	0	0/12	0/12
Low-performers	M=2.4, SD=3.4	M=.6, SD=.9	M=15, SD=14	4/11	3/11
Mid-performers	M=3.8, SD=5.7	M=.1, SD=.4	M=22, SD=22	1/8	1/8
High-performers	M=6.6, SD=6.1	M=.8, SD=1.1	M=50, SD=37	5/9	5/9

right section of Table III. The only break in the upward trend was in the mid-performer group, where only one participant deleted one *Do together* or *Repeat loop* and only one nested them. This may be due to small sample size, especially within the mid-performer group. Regardless, deletion and nesting of *Do together* and *Repeat loop* code blocks is consistent across groups. This may mean that these two behaviors relate to the types of skills that result in successful knowledge application.

Novices' interactions with code blocks may indicate long before a user has completed their program how much they know or how much knowledge they will gain. This may mean that tools for **open-ended programming** can begin to adapt to the personal needs of the programmer before they have even completed one program. Current tools provide support and encouragement for novices to use certain types of code blocks, but not **how to interact with** those code blocks in ways that might support deeper understanding.

VII. DISCUSSION

Future support for open-ended novice programming may be able to personalize support throughout novices' programming experience based on their interactions across the programming environment and support tools. The relationship between deep interaction with support tools and exploration also suggests that future tools should support non-explorers.

A. Behavior-based support

The connections between users' behaviors and successful knowledge application indicates that we may be able to predict skill level or likelihood to learn new concepts before a user has completed their project. This prediction could enable a tool to provide a user with the amount and type of support that will benefit them the most. The correlation between deep interaction with suggestions and exploration of the interface may suggest that those who explore on their own benefit from suggestions in their current form. Those who do less exploring on their own might benefit from other types of support which do not require the user to explore. Adapting support based on perceived user needs has risks, particularly in irritating users. Triangulating users' interactions with the interface, code and support tools may enable accurate identification of the users who will benefit the most from support for non-explorers.

B. Support for non-explorers

Self-directed open-ended programming requires the user to create their own knowledge. At least some subset of the population typically does not explore on their own. Still, little to no support has been developed within these contexts for those who do not explore naturally on their own. The suggestions in this study have been one of the first systems to explicitly and successfully encourage novices to explore new code during open-ended programming. Participants of all types seem to access suggestions, but access alone likely isn't enough to understand more complex concepts. Requiring the user to execute the code snippet or switch between examples forces the user to have to 'explore' the suggestion in order to benefit from it.

Non-explorer open-ended programmers may benefit from tools designed explicitly with the non-explorer in mind. For example, these tools may: appear in relevant locations, 'force' deeper interactions, or demonstrate exploring behaviors. A tool could select where to display support based on where the novice programmer focuses the most. In our study, those who performed better spent more time exploring code. Those who performed worse spent more time executing code. Suggestions and tutorials were both displayed in a scroll-able list right above the code menu, which likely seemed very similar to the code list. Tools for open-ended novice programming might benefit from providing more guidance through support interactions. Those who explored more tended to leverage the deeper interactions within suggestions and tutorials compared to those who explored less. An opened suggestion or tutorial could, for example, require a non-exploring user to step through the deeper interactions before closing it. Typical support often provides static information, rather than instruction on how to explore. Suggestions and tutorials provide examples of how to create specific animations. Suggestions nor other support tools, commonly provide instruction or encouragement to spend more time trying out how the code works. Tools directed at non-explorers might actually demonstrate the types of exploring behaviors that other users already choose to perform on their own.

VIII. CONCLUSION

This paper has two contributions: 1) the first exploration into novices' purely open-ended programming behaviors and how those relate to knowledge application, and 2) two evidence-based directions for support for open-ended programming. Overall, we found that those who performed better often spent more time exploring the available code and interacting with code blocks. Deeper interactions with support tools also had a relationship with exploration. We presented two implications for designing support for open-ended, self-directed novice programmers. Our results indicate that novices' behaviors during programming may provide valuable clues for adaptive tools. Further, the design of future tools should likely focus on how to support those novice programmers who do not naturally explore on their own.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1054587 and 1440996.

REFERENCES

[1] Efthimia Aivaloglou and Felienne Hermans. 2016. How kids code and how we know: An exploratory study on the Scratch repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 53–61.

[2] Demis Basso, Ilenia Fronza, Alessandro Colombi, and Claus Pahl. 2018. Improving Assessment of Computational Thinking Through a Comprehensive Framework. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. ACM, 15.

[3] Laura Beckwith, Margaret Burnett, Valentina Grigoreanu, and Susan Wiedenbeck. 2006. Gender HCI: What about the software? *Computer* 39, 11 (2006), 97–101.

[4] Karen Brennan and Mitchel Resnick. 2012. New frameworks for studying and assessing the development of computational thinking. 1 (2012), 25.

[5] Stephen Cooper, Wanda Dann, and Randy Pausch. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges* 15, 5 (2000), 107–116.

[6] Leonel Morales Diaz, Laura S Gaytan-Lugo, and Lissette Fleck. 2015. Profiling styles of use in Alice: Identifying patterns of use by observing participants in workshops with Alice. (2015), 19–24.

[7] Yihuan Dong, Samiha Marwan, Veronica Catete, Thomas Price, and Tiffany Barnes. 2019. Defining Tinkering Behavior in Open-ended Block-based Programming Assignments. (2019), 1204–1210.

[8] Paul Gross and Caitlin Kelleher. 2010. The Looking Glass IDE for learning computer programming through storytelling and history exploration: conference workshop. *Journal of Computing Sciences in Colleges* 26, 1 (2010), 75–76.

[9] Kyle James Harms, Jason Chen, and Caitlin L. Kelleher. 2016. Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 241–250.

[10] Felienne Hermans, Kathryn T Stolee, and David Hoepelman. 2016. Smells in block-based programming languages. In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*. IEEE, 68–72.

[11] Wint Hnin, Michelle Ichinco, and Caitlin Kelleher. 2017. An Exploratory Study of the Usage of Different Educational Resources in an Independent Context. In *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE, To Appear.

[12] Michelle Ichinco, Wint Yee Hnin, and Caitlin L. Kelleher. 2017. Suggesting API Usage to Novice Programmers with the Example Guru. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 1105–1117.

[13] Michelle Ichinco and Caitlin Kelleher. 2018. Semi-automatic suggestion generation for young novice programmers in an open-ended context. (2018), 405–412.

[14] Barry Peddycord Iii, Andrew Hicks, and Tiffany Barnes. 2014. Generating hints for programming problems using intermediate output. In *Educational Data Mining 2014*. Citeseer.

[15] Will Jernigan, Amber Horvath, Michael Lee, Margaret Burnett, Taylor Cuilty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, and Andrew Ko. 2015. A principled evaluation for a principled Idea Garden. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 235–243.

[16] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1455–1464. http://dl.acm.org/citation.cfm?id=1240844

[17] J Nathan Matias, Sayamindu Dasgupta, and Benjamin Mako Hill. 2016. Skill progression in scratch revisited. (2016), 1486–1490.

[18] Jesús Moreno-León, Gregorio Robles, and others. 2015. Dr. Scratch: a Web Tool to Automatically Evaluate Scratch Projects. (2015), 132–133.

[19] Jesús Moreno-León, Marcos Román-González, Casper Harteveld, and Gregorio Robles. 2017. On the automatic assessment of computational thinking skills: A comparison with human experts. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 2788–2795.

[20] Jean Piaget. 1972. Intellectual evolution from adolescence to adulthood. *Human development* 15, 1 (1972), 1–12.

[21] Shaileen Crawford Pokress and José Juan Dominguez Veiga. 2013. MIT App Inventor: Enabling personal mobile computing. *arXiv preprint arXiv:1310.2830* (2013).

[22] Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 483–488.

[23] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S Silver, Brian Silverman, and others. 2009. Scratch: Programming for all. *Commun. Acm* 52, 11 (2009), 60–67.

[24] Mary Budd Rowe. 1978. Teaching science as continuous inquiry. (1978).

[25] Christopher Scaffidi and Christopher Chambers. 2012. Skill progression demonstrated by users in the Scratch animation environment. *International Journal of Human-Computer Interaction* 28, 6 (2012), 383–398. http://www.tandfonline.com/doi/abs/10.1080/10447318.2011.595621

[26] Scratch - Imagine, Program, Share 2016. (2016). https://scratch.mit.edu/help/videos/ Accessed: 2016-04-01.

[27] Snap 2019. Snap! (Build Your Own Blocks) 4.2. (2019). https://snap.berkeley.edu/

[28] Alaaeddin Swidan, Alexander Serebrenik, and Felienne Hermans. 2017. How do scratch programmers name variables and procedures?. In *Source Code Analysis and Manipulation (SCAM), 2017 IEEE 17th International Working Conference on*. IEEE, 51–60.

[29] Peeratham Techapalokul and Eli Tilevich. 2017a. Quality HoundAn online code smell analyzer for scratch programs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE, 337–338.

[30] Peeratham Techapalokul and Eli Tilevich. 2017b. Understanding recurring quality problems and their impact on code sharing in block-based software. In *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE, 43–51.

[31] Christiane Gresse von WANGENHEIM, Jean CR Hauck, Matheus Faustino Demetrio, Rafael Pelle, Nathalia da CRUZ ALVES, Heliziane Barbosa, and Luiz Felipe Azevedo. 2018. CodeMaster-Automatic Assessment and Grading of App Inventor and Snap! Programs. *Informatics in Education* 17, 1 (2018).

[32] Benjamin Xie and Hal Abelson. 2016. Skill progression in MIT app inventor. (2016), 213–217.