Instr Sci
https://doi.org/10.1007/s11251-018-9453-5

ORIGINAL RESEARCH

Debugging during block-based programming

ChanMin Kim[1] · Jiangmei Yuan[2] · Lucas Vasconcelos[1] · Minyoung Shin[1] · Roger B. Hill[1]

Received: 5 February 2018 / Accepted: 2 April 2018

Abstract In this study, we investigated the debugging process that early childhood preservice teachers used during block-based programing. Its purpose was to provide insights into how to prepare early childhood teachers to integrate computer science into instruction. This study reports the types of errors that early childhood preservice teachers commonly made and how they debugged the errors. Findings are discussed in relation to research and practice that could benefit from debugging instruction. This study provides directions for future computer science education research that aims to prepare teachers for programming, computational thinking, and STEM education. Though this study used robotics as a programming context, findings on early childhood preservice teachers' debugging processes could be applicable to other contexts involving block-based programming.

Keywords Early childhood education · Computer science education · Teacher preparation · Block-based programming · Debugging · Educational robotics

Introduction

The USA initiative entitled *Computer Science for All* emphasizes that everyone needs to learn to program (K12 Computer Science Framework Steering Committee 2016). However, little is known about how to prepare early childhood teachers for such an initiative. Learning core programming concepts and processes (e.g., abstraction, conditional logic, and recursion) does not require learning text-based programming languages like Python or C++; rather, it is possible using the more approachable block-based programming languages. Block-based programing has been used successfully in early childhood education (Bers et al. 2014). But to be used more widely in early childhood education, preservice

✉ ChanMin Kim
chanmin@uga.edu

[1] College of Education, The University of Georgia, Athens, GA, USA

[2] College of Education and Human Services, West Virginia University, Morgantown, WV, USA

Published online: 12 May 2018

early childhood teachers need to gain the skills and confidence to engage in block-based programming. Critical to this is learning how to debug. In the present study, we examined the block-based programing process of early childhood preservice teachers, specifically focusing on what bugs occurred and how they were resolved. This is a first step toward understanding how to support preservice early childhood teachers to learn to engage in block-based programming, and thus have the confidence and skill to support such activity among their future students.

Our focus was on their debugging process because (a) a successful output of block-based programming (i.e., code that functions) does not necessary equate to successful learning of programming concepts and processes (Brennan and Resnick 2012; Grover et al. 2015), (b) the examination of debugging processes could inform partial understanding and misunderstanding of programming (Perkins and Martin 1985), and (c) debugging is critical in computer science (McCauley et al. 2008), problem solving (Jonassen 2000; Yen et al. 2012), and computational thinking (Brennan and Resnick 2012; Committee for the Workshops on Computational Thinking and National Research Council 2010).

Relevant literature

Block-based programming and debugging

Compared to text-based programming that requires syntax, block-based programming is an inviting way to engage novice learners in programming and help them learn abstract computer science concepts (Akcaoglu 2014; Bers et al. 2014; Lye and Koh 2014). Because of this, block-based programming is often paired with educational robotics such that learners can control robots' movement to accomplish a task (e.g., Bers, 2010; Bers et al. 2013; Kazakoff and Bers 2012).

Despite the potential benefits of block-based programming, learner engagement in fundamental programming is often shallow and unsuccessful (Grover et al. 2015). For example, functioning code does not necessarily demonstrate that the programmer understands how the code works; rather, the code may work due to successful remixing of reusable code through tinkering (Brennan and Resnick 2012). Critical to the promotion of mindful engagement in fundamental programming concepts is debugging, defined as the systematic process of identifying why code malfunctions and fixing the cause. Debugging is central to programming and computer science, and part of problem solving (Jonassen 2000; McCauley et al. 2008; Yen et al. 2012) required in STEM and beyond.

Teachers as debuggers

Helping teachers gain skill and confidence in debugging is critical to helping them teach students to debug. Learning to debug while learning to program improved programming concept understanding and programming performance in the context of text-based programming (Chiu and Huang 2015; Lee and Ko 2015). Learning to debug also can help with problem solving in general (Yen et al. 2012) because debugging involves problem (i.e., bugs) identification and solution generation (McCauley et al. 2008), which is the essence of problem solving. Problem-solving skills are necessary in the twenty first century work and living (Greiff et al. 2014). It is especially important to learn to solve ill-structured problems with multiple solution paths (Jonassen 2011).

Debugging can be difficult to even professional programmers because it requires mindful, persistent engagement. Tinkering and simple removal of a buggy portion are frequent methods of debugging (Kim et al. 2017; Bers et al. 2014; Jadud 2006). These methods are problematic because they can result in functioning code but fail to inform the debugger of the origin of errors and resolution. Without knowing the origin of errors, future teachers will be unable to teach students why and how things work. Another problem observed in debugging research during block-based programming is that many early childhood preservice teachers were afraid to make errors, and thus chose an easier, simpler path to program a robot (Kim et al. 2016). In Kim et al. (2016), preservice teachers who made such a choice learned less than preservice teachers who challenged themselves by having to study advanced programming. These studies suggest that there is a need for focused research on debugging in block-based programming to help teachers (and thereby their students) learn to perform reliable debugging (e.g., logic debugging).

There has been much research on novice programmers' debugging during text-based programming (McCauley et al. 2008). However, it is not clear how such research is applied to block-based programming. There has been a recent effort to teach middle and high school students to debug during block-based programming for game-design (Akcaoglu 2014; Chiu and Huang 2015) but there is little detail on debugging strategies used. Specific debugging steps (e.g., hypothesis generation) were examined during robot programming of kindergarten students in Bers et al. (2014) but not in a teacher education context.

Research purpose and questions

Given the gaps discussed above, we investigated (a) the errors (i.e., bugs) early childhood education preservice teachers made during block-based programming and (b) how they debugged the errors. We expected such investigations would inform how preservice teachers could and should be prepared to be a debugger who can educate young children to solve problems. The following research questions were addressed:

1. What are common errors (i.e., bugs) that early childhood education preservice teachers made during block-based programming?
2. How do they debug the errors?

Methods

Research design

A case study design (Leedy and Ormrod 2013) was used to provide an in-depth understanding of the errors and the debugging processes during robot programming. Actions and conversations during and about programming were investigated to identify patterns (e.g., common programming errors) and themes (e.g., infrequent use of backward reasoning in locating bugs) that characterize these early childhood education preservice teachers' programming errors and debugging processes.

Research context and participants

This study involved 19 participants who attended a course designed for students majoring in early childhood education during the 2016 spring semester offered at a public university in the southeastern region of the USA. This course invited students to learn hands-on teaching skills that required manipulation of various technologies. All participants were female students enrolled in an Early Childhood Education major. Two participants were excluded from the data analysis. One, Avery (note: all names are pseudonyms), indicated in her interview that her team had encountered errors a few times but did not elaborate on the team's debugging processes other than summarizing them as "just a lot of trial and error". The computer screen of Avery's team was not recorded, and their classroom activities were video-recorded but the video segment including programming was accidently deleted. The other, Harper, noted in her interview that she had experienced no error during programming but the classroom activity video and the computer screen recordings showed that there was a coding error. However, the debugging process could not be analyzed due to a lack of related information—Harper neither spoke nor debugged in the video. The average age of the 17 participants included in the data analysis was 20.29 (SD = 2.05). All but one were White (94.12%). One was black (5.88%). Before joining this study, 13 (76.48%) participants had completed at least three semesters in college, and 2 (11.76%) had one to two semesters completed. Two (11.76%) were attending their first semester.

Robotics learning module

The robotics learning module was three-weeks long (two 2-h class meetings per week), and followed the format of (a) individual practice with robot assembly and programming, (b) group work in which students assembled and programmed a robot and designed a lesson using the robot, and (c) poster presentation about their team robot and lesson. Block-based programming was used. Figure 1 shows a robot that a student group assembled and programmed, and used in their lesson.

Procedures

Participants were recruited in class before the 3-week module for robotics learning began. Researchers visited the class in person. Before and after the robotics learning module, the

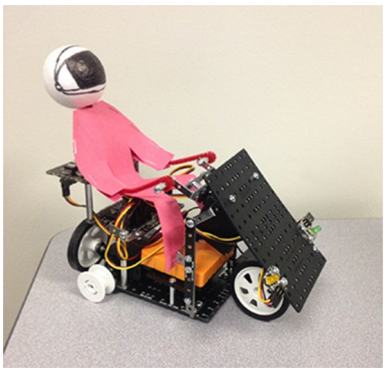

Fig. 1 An example robot

surveys were conducted. Interviews were done after the post survey responses were collected. For 3 weeks during two 2-h class meetings per week, participants learned to assemble and program robots, individually first and then in a team. Their end-of-module products were a poster presentation about their robot and a lesson using the robot.

Data collection

Data were collected from classroom recordings, computer screen recordings, and interviews. Classroom recordings were done for each of 12 teams. The total length of videos for each team was 4–5 hours. Computer screens of six teams were recorded using Screencast-O-Matic (see Fig. 2). The length of each team's screen recording was 60–80 minutes. Fifteen out of 19 participants volunteered for an individual interview. Interviews were semi-structured with questions investigating difficulties that participants encountered and their problem-solving processes during robot building and programming (e.g., Tell us about challenges during programming; How did you overcome the challenge? Did you have to make a choice to give up and move on?). Part of the interviews included buggy block-based code to prompt memories of errors and debugging processes. The length of interviews ranged between 20 and 30 minutes.

Data analysis

To address Research Question 1, we used open coding. We limited our analysis to errors related to programming. For example, we did not code conversation excerpts related to assembly (e.g., conversations about loose screws). To address Research Question 2, we used predetermined coding schemes based on (a) Vessey's (1985) debugging paths (i.e., examined output or program first? → Active or passive examination? → Constrained by

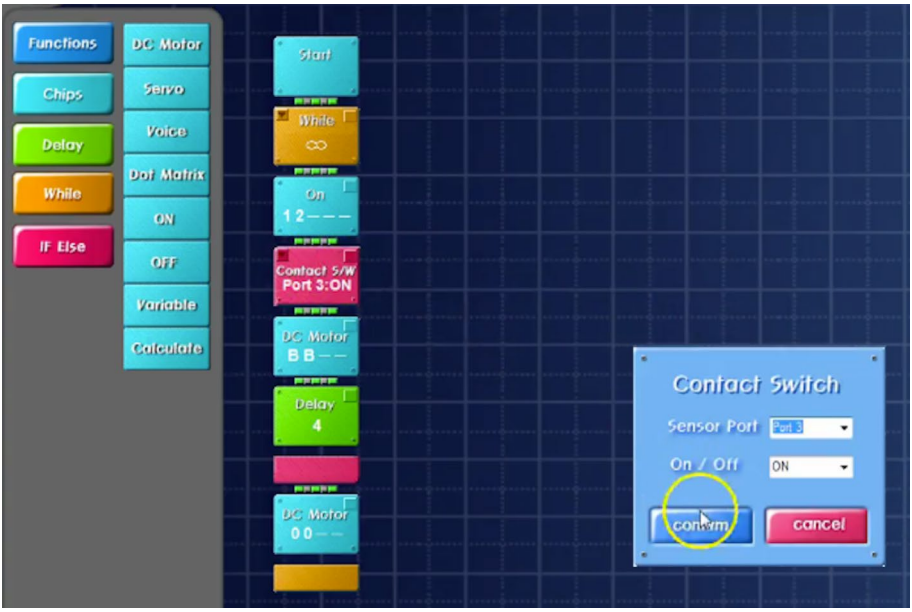

Fig. 2 Example of video-recorded computer screen during programming

hypothesis? → Developed a model of the program structure? → Deduced a causal model of the error?) and (b) Katz and Anderson's (1987) error-locating techniques (i.e., working backwards, in program order, and in serial order). In both processes, we first identified actions during programming and conversations about programming to analyze only programming-relevant data. The second, third, and fourth authors were each responsible for coding one third of the data, and then each discussed the coding decisions and resulting analyses with the first author to arrive at consensus.

Findings and discussion

RQ1: what are common errors (i.e., bugs)?

There were six common errors in programming (see Table 1). First, participants *defined incorrect variable values*, such as defining the incorrect speed of a motor for a robot to perform a turn (see Fig. 3). For example, Claire reported, "I could not get it to make the square perfectly… even though it was in decimals. It'd be like, 'Turn for 3.215 seconds,' but it was still not good enough. Like not accurate enough to make it turn quite right." Second, participants *selected motors* (*port*) *that were incongruent with the physical object* (*the robot*). For example, Motor 1 was connected to the right arm of the robot but the motor selected in programing was Motor 2. Third, participants made *errors in locating the right composition of the motor and designating the right speed*. That is, a set of motors, such as two motors for two arms each, oftentimes required different speeds to perform the same action, which led to many errors to get to the right composition of the motor and its speed for each arm. Fourth, participants *omitted commands that were necessary for the robot to perform as planned*. For example, robots moved constantly or did not move at all because of omitting loop or other commands that had to be included to complete the program. Fifth, participants *improperly defined conditionals*. For instance, participants had difficulty programming the result of pressing a remote control button. Last, participants *searched for programming errors when a malfunction was due to other factors*. Participants kept searching for an error in the code that they were programming when the error was due to assembly rather than programing.

The error list can be used in teaching debugging strategies. For example, a checklist showing these common errors can be given to preservice teachers whose robot does not work as planned to help them generate and test a hypothesis for debugging. Considering the importance of generating a hypothesis in debugging (Araki et al. 1991), such a list could improve these novice programmers' learning to debug. Ahmadzadeh et al. (2007) argued that common errors should be integrated into debugging exercise within college introductory programming courses because "each error plays a significant role in the process of learning" (p. 73) to program. However, the goal should not be to prevent struggle in identifying bugs, in that preservice teachers need to experience and resolve productive struggle to be able to help their future students to do the same (Kim et al. 2016). For example, Chiu and Huang (2015) used worksheets designed around common errors in Scratch to guide debugging but step-by-step debugging strategies were listed on the worksheets for each error. Such worksheets like job aids may have been necessary in their study context. However, the methods of using common errors in teacher preparation need to include scaffolding, defined as temporary and customized support that helps one solve problems and leads to his or her acquisition of necessary skills for problem solving (Wood et al. 1976),

Table 1 Common programming errors in the present study

Common errors	Interview excerpts illustrating the errors
(1) Error in defining a value in variables	I could not get it to make the square perfectly… even though it was in decimals. It'd be like, "Turn for 3.215 seconds," but it was still not good enough. Like not accurate enough to make it turn quite right (Claire)
(2) Error in selecting a motor (port) that is consistent with the physical object (the robot)	He [the instructor] said the board was not … connected it to Port 7 (Taylor)
(3) Error in locating the right composition of the motor and its speed	Um, I was kind of a little out of time because one of our motors was way faster than the other one, so when it would go straight, it wouldn't actually go straight. It'd veer off, so we had to keep adjusting it to make it um, slow down one motor and speed up one motor so it would actually go straight (Taylor)
(4) Error of omitting a command that is necessary for the robot to perform as planned	Yeah, I would, um, create this program, and he [the instructor] eventually taught me that it was because it wasn't in a loop or the while loop, that's why it wasn't moving. I would download it, and I would go to start it, and it wasn't even moving, so I would just, I was so confused, and that's when he had to come over and say, "Oh no, you need it in a while loop or some kind of loop (Aubree)
(5) Error in defining conditions	Making sure that the remote, you know, if you pressed this, then this happens. But if you don't press it, then this happens. So like the if/then aspect of programming was kind of difficult. And in all activities, the if/then was kind of hard for me to follow (Giana)
(6) Error in recognizing the program *without* an error	Every time we would turn our robot on after downloading the program, it would automatically turn sideways and stick straight up, and we couldn't figure out why. And then finally, we realized that on the motor, it wanted to start in a specific position, where you plug in the wheel or whatever on the motor. It wanted to start in a certain way, and we just kept moving it back. So once we figured that out, then we realized the handlebars wouldn't fit onto that point the way it wanted to, so we kind of had to modify how the handlebars were built and then get them on (Abbie)

rather than job aids (Belland 2014). Scaffolding preservice teachers to debug will likely result in their scaffolding students to debug because teaching practice generally resembles the methods used when teachers were taught.

Half of the common errors (i.e., #1, 4, and 5) in the present study were similar to those in other studies on programming *without block-based programming* or *without robots*. The use of conditionals and loops were major errors consistently found in studies using text-based programming (Ahmadzadeh et al. 2007; Fitzgerald et al. 2008; Liu et al. 2017) and block-based programming without robots (Chiu and Huang 2015). For example, in Ahmadzadeh et al. (2007), conditionals and loops were the top two of which introductory programming students made the most errors regardless of having debugging exercise or not. Further research is needed to analyze similarities and differences between common

Fig. 3 Interface for selecting a motor and defining a variable

errors discussed in literature on text-based programming and block-based programming without physical objects (Klahr and Carver 1988; e.g., McCauley et al. 2008) and those found in the present study. Such research could guide effective ways of teaching debugging in block-based programming.

RQ2: How do preservice early childhood teachers debug the errors?

Table 2 lists a summary of each participant group's debugging processes analyzed using coding schemes based on (a) Vessey's (1985) debugging paths and (b) Katz and Anderson's (1987) error-locating techniques (see details in "Data analysis" section).

Output was examined first

The output on a computer screen in text-based programming is not the same as the output in robot programming in the form of a physical object—robot behaviors. However, in the sense that the output is a result of running code, the output in the two different programming interfaces is the same. In the present study, the majority of debugging processes began with examining the *output* (i.e., the robot), not the *program* (i.e., block-based code). For example, the team of Taylor and Aubree wanted their robot to run in a straight line and then make a 180° turn, but the robot ran only half way through the straight line and made a turn of more than 180°. When noticing the errors, they immediately examined the robot to determine the source of the problem. It seems natural that preservice teachers started their debugging process by examining the output since they were familiar with the robot after multiple assembly opportunities. In fact, after going through multiple debugging processes that began with examining the output first, some participants examined the program first. For example, the team of Abbie and Emma went through two debugging processes examining the robot first before they started looking at the program first. It appears that after

Table 2 A sample of data analysis on debugging process

	Debugging Process: Step 1		Debugging Process: Step 2		Debugging Process: Step 3			Debugging Process: Step 4				Error-locating technique		
Team members	Examined program first	Examined output first	Active examination	Passive examination	Constrained by one hypothesis	Not constrained by one hypothesis	No hypothesis	Developed a model of the program structure	Did not develop a model of the program structure	Deduced a causal model of the error	Did not deduce a causal model of the error	Work forward in program order	Work forward in serial order	Work backward
Leah, Claire		✓	✓				✓	✓			✓	✓		
Leah, Claire		✓	✓				✓	✓			✓	✓		
Taylor, Aubree		✓		✓			✓	✓			✓			
Taylor, Aubree		✓		✓			✓	✓			✓			
Taylor, Aubree	✓		✓	✓	✓			✓			✓			✓
Sara	✓			✓			✓	✓			✓			
Abbie, Emma		✓	✓			✓		✓		✓				
Abbie, Emma		✓	✓			✓		✓		✓				
Abbie, Emma	✓		✓			✓		✓		✓				
Paula, Olivia		✓	✓				✓	✓			✓			
Anna, Emily	✓			✓	✓			✓			✓		2	1

Table 2 (continued)

Team members	Debugging Process: Step 1		Debugging Process: Step 2		Debugging Process: Step 3			Debugging Process: Step 4				Error-locating technique		
	Examined program first	Examined output first	Active examination	Passive examination	Constrained by one hypothesis	Not constrained by one hypothesis	No hypothesis	Developed a model of the program structure	Did not develop a model of the program structure	Deduced a causal model of the error	Did not deduce a causal model of the error	Work forward in program order	Work forward in serial order	Work backward
Anna, Emily	✔		✔		✔			✔		✔			2	1
Alante		✔	✔		✔			✔		✔		1	2	
Lindi, Kaylee		✔	✔	✔		✔		✔			✔			✔
Giana, Maya		✔		✔			✔		✔		✔		✔	
Lori		✔		✔			✔	✔			✔		✔	

Pseudonyms are used. This table shows only a summary sample. Participant rows with only one name are from teams with only one research participant. Rows are organized per debugging process, which is why there are multiple rows for some participants. Steps 1 through 4 columns list results of data analysis using coding schemes based on Vessey's (1985). Error-locating technique column lists results of data analysis using coding schemes based on Katz and Anderson's (1987). Numbers in the error-locating technique cells indicate the order of techniques used

participants became familiar with the program after going through multiple debugging processes, they examined the program first.

The theme of examining output first can be attributed to the comprehension phase in which the majority of participants may have been—in search of "the difference between the actual execution and anticipated outcomes" (Yen et al. 2012, p. 124). Although participants' observation of their robot's unanticipated behavior (including no behavior) prompted them to conduct an examination, they may have still been in the process of identifying how different the robot's actual behavior was from the anticipated one. This appears aligned with Jadud's (2005, 2006) findings of novice Java programmers' frequent recompiling—the process of wondering "exactly what error they were about to attempt to fix" (Jadud 2005, p. 35). In another study on Java programming (Ahmadzadeh et al. 2005), novice programmers who failed to debug lacked an understanding of the actual program behavior.

This theme seems related to the findings on hypothesis generation (see more in "Successful debugging (i.e., fixing the error) did not mean a deduction of a causal model" section). That is, mostly when participants examined the output first, no hypothesis for the cause of the error was generated during the debugging process (e.g., all of Leah and Claire's debugging processes and Taylor and Aubree's first two debugging processes), and mostly when participants examined the program first, a hypothesis was generated (e.g., Taylor and Aubree's third debugging process). A large consensus among debugging researchers is that there are two overall phases of the debugging process—comprehension and correction (Yen et al. 2012). Between the two phases, the comprehension phase of participants in the present study may have not been fully completed considering that (a) the correction phase comes after the comprehension phase and (b) "the isolation strategy (or fault diagnosis strategy)" (Yoon and Garcia 1998, p. 161) during the correction phase involves hypotheses. In other words, the incomplete comprehension phase can be inferred from having no hypothesis. It is not impossible for debuggers to skip the comprehension phase (Yen et al. 2012) but when they are unfamiliar with the program (e.g., when debugging someone else's program or when the debugger is a novice programmer), the comprehension phase must happen (Tsau 1996).

One exception to the theme of examining the output first was the team of Anna and Emily who examined the program first in all debugging processes. Sara also examined the program first but the instructor explained to her the origin of the error. For example, Anna and Emily examined the program immediately after they saw the robot switch's malfunction. They seemed familiar with the program from the beginning considering other parts of their debugging process: they always had a hypothesis for the cause of the error, which means they knew enough to move onto the correction phase (Tsau 1996; Yen et al. 2012; Yoon and Garcia 1998). Further research on preservice teachers like Anna and Emily who always examined the program first could lead to the knowledge of what is needed to help the rest learn to examine the program first.

Learning to examine the program first may lead to effective debugging. However, to do so, participants would appear to need scaffolding during the comprehension phase. The list of common errors could be used in designing conceptual scaffolding to help not only error diagnosis (i.e., the correction phase) but also identifying discrepancy between the planned and actual robot's behaviors (i.e., the comprehension phase). Conceptual scaffolding refers to scaffolding that "guides students in terms of things to consider when solving problems… to help students narrow these down and choose more productive considerations" (Belland, 2017, p. 109). For example, when the robot ran only half way through a straight line, participants Taylor and Aubree could (a) be scaffolded to consider the six common errors, (b) choose to examine the value entered for the number of loop in the program (the common

error in defining a value in variables in "RQ1: what are common errors (i.e., bugs)?" section), and (c) identify that the discrepancy between their planned and actual robot's behaviors was from their miscalculation of the number of loops needed to complete the robot's travel in a straight line due to an incorrect calculation of the robot wheel circumference.

Most debugging processes involved active examination

Most debugging processes involved active examination including multiple reviews and testing of the program. For example, when their robot did not move as planned, Abbie and Emma had the robot perform again, looked into the robot and the code back and forth several times, and then examined the code. Paula and Olivia's active examination involved referring to multiple sources such as the robot manual, the robot, and the code as indicated in Olivia's following comment: "The only problem we encountered was [that] we couldn't exactly remember how to make it [the robot] move forward. I knew I needed a delay, and I just couldn't remember which buttons [blocks]… I just would refer back to the code we had already written to figure out what we needed to do."

There were a few processes in which participants passively received help from the instructor or a peer. Taylor and Aubree asked the instructor for help immediately after seeing that their remote control did not work. The instructor explained how to fix the remote control, and they began to test the robot. The robot did not go straight, and the instructor, who was still with them, told them that there could be two possible causes for the problem and two ways to fix the problem. They chose one of the ways, and changed the speeds of the two motors in a trial-and-error manner. In these passive processes, mostly no hypothesis was formed. In one case (Anna and Emily), a hypothesis was formed but it was hinted by the instructor upon the participants' request. In Gould (1975), programmers generated a hypothesis when (a) a suspicious line was detected but they had little clue about how to respond to the suspicious line, or (b) no suspicious line was detected. In the present study, the participants who engaged in passive examination did not search for a suspicious block (*block* instead of *line* due to block-based, not text-based, programming in the present study), which in turn led to no hypothesis.

There were participants who began with active examination for possible bugs but ended up with passive examination—the team of Taylor and Aubree and the team of Lindi and Kaylee. For example, the programming goal of Lindi and Kaylee was to have their forklift robot lift the fork up and take it back down a few times but the fork did not go back down. They explored more than one hypothesis. For example, they checked the motor and changed chunks of program codes to make the robot fork move up and down. However, they had no success and switched to passive mode, and the instructor ended up pointing out the section in the robot kit manual that indicated how the code could be fixed. Research shows that novice programmers tend to perform low quality hypothesis testing more often than skilled programmers (McCauley et al. 2008), and invite new bugs during such testing (Gugerty and Olson 1986; McCauley et al. 2008). Considering that locating bugs is more difficult than correcting them in general (Fitzgerald et al. 2010; Katz and Anderson 1987), methods of scaffolding them to complete the bug search need to be studied so that preservice teachers like Lindi and Kaylee who began with active examination do not give up on finding a bug. Structuring task components that are not central to locating a bug and problematizing task components that are central to locating a bug (Reiser 2004) may help participants stay on task. Lindi's comment, "I got really frustrated when it didn't work, and you could probably see that in the [classroom recording] videos", hints

that the debugging process became overwhelming and frustrating. Scaffolding specifically for productive struggle, defined as the process of attempting to overcome difficulties during complex problem solving through exerting persistent effort toward sense-making (Hiebert and Grouws 2007), could be helpful. For example, strategies to reduce uncertainty such as "check for likely errors" (Schunn and Trafton 2013, p. 474) could help these preservice teachers struggle through the challenge like "uncertainty in explaining and sense-making" (Warshauer 2014, p. 11).

In sum, all passive processes led to no causal model of the error; that is, participants did not know what caused the error and how the error was fixed. A robotics learning module requiring preservice teachers to attempt to discover an erroneous block for a certain amount of time could facilitate active examination during a debugging process.

Backward reasoning was infrequently used in locating errors

When locating an error, participants used forward reasoning (i.e., reviewing the program in program order or in serial order) more than backward reasoning. Although the output was examined first in general (see "Output was examined first" section), when participants reviewed the program to locate errors, few participants worked backwards. For example, Leah and Claire programmed the robot to run along a square but the robot failed to make a perfect square because it turned too much or too little. They went through each of the blocks related to the robot's turning one by one from the beginning. They used a causal reasoning strategy that "involves looking at the information obtained in the testing of the function … reasoning about what might be causing the bug … reading over the program (simulating) from the beginning…" (Katz and Anderson 1987, p. 365). Leah and Claire reported having "to keep playing with different buttons [blocks] and figuring out. I made the one wheel turn faster than the other one when I made it turn." Their error-locating technique in this incident was "forward reasoning program order (related to simulating the program's execution)" (McCauley et al. 2008, p. 77). In cases of "forward reasoning serial order (the order the lines [blocks] appear as the code is read)" (McCauley et al. 2008, p. 77), for example, Abbie and Emma reviewed the blocks in the program from top to bottom block by block. There were more cases of forward reasoning serial order than cases of forward reasoning program order.

These findings may be related to participants' unfamiliarity with programming (Katz and Anderson 1987; McCauley et al. 2008), considering that the robotics learning module was their first programing experience. "To effectively use a backward-reasoning strategy, knowledge of why an algorithm was implemented in a particular way is necessary. In contrast, having primarily knowledge of how a program is executed may lead to a forward-reasoning strategy" (Katz and Anderson 1987, p. 387). However, error-locating techniques are trainable. For example, programmers used the technique that they were trained to use even when they had autonomy to choose in Katz and Anderson (1987). Preservice teachers may benefit from explicit training to use backward reasoning to locate an error since the technique tends to lead to a more accurate solution than forward reasoning (Katz and Anderson 1987; McCauley et al. 2008). However, familiarity with the code is needed for backward reasoning (Yen et al. 2012). Thus, to establish familiarity, mappings between bugs and robot behaviors (including no behavior) could be compiled and reasoned (Klahr and Carver 1988), and practiced among these preservice teachers.

Considering that locating bugs is often more difficult than fixing them (Ahmadzadeh et al. 2005; Fitzgerald et al. 2010; McCauley et al. 2008), teaching preservice teachers about techniques for isolating bugs could improve their debugging, and programming

overall. Researchers have highlighted the need for novice programmers to differentiate locating bugs from fixing bugs (Carver and Risinger 1987; Simon et al. 2008). Also, considering that studies with novice programmers observed that being a skillful programmer does not necessarily mean being a skillful debugger, teaching programming skills without addressing debugging skills would not be ideal.

Successful debugging (i.e., fixing the error) did not mean a deduction of a causal model

Not all participants who fixed an error knew what caused the error and how it was resolved. For example, Claire's interview comments illustrate that a causal model of the error was missing even after successfully correcting the error:

> I don't know exactly what the problem was. … maybe one wheel goes faster than the other…. Maybe if all of the parts of the robot aren't screwed in the same amount …. Maybe that's what makes it… a little bit different than the other. I don't know. (…) I had to keep playing with the different buttons [blocks] and figuring out. Like I made the one wheel turn faster than the other one when I like made it turn. (…) So the one wheel, like, okay for example, when I made it turn, the left wheel of the robot, I've set it on five, like the speed of five, and then the right wheel I made it go backwards, instead of making it go backwards at five, I made it go backwards at four.

This finding may have resulted from novice programmers' tendency to focus on a solution itself rather than how the program works (Vessey 1985). The debugging processes with passive examination (discussed in "Most debugging processes involved active examination" section) depicts this tendency—when given the bug location information, participants hardly asked the instructor how the erroneous portion caused the robot problem; rather, they ended up receiving the debugging method information as well. For example, when the robot of Sara's team could not raise its arms to pierce balloons, the instructor explained to the team that the source of the error was the robot's servo motor. Although the team's changes in the servo motor setting corrected its arm movement, Sara was not sure about the exact relation between the servo motor, the numerical value entered in the code, and the robot arms' movement. That is, she arrived at a solution, "which numbers were best to use", as she commented, but not at a causal model of error. This also exhibits that understanding the program structure did not always lead to a deduction of a causal model of the error. That is, Sara learned that the motor affected the robot's arms' malfunctioning (i.e., understanding the structure of the program) when the instructor told her that the motor setting in the program must be wrong. However, even after the error was fixed, Sara was not sure how it was fixed other than finding a correct speed (the best numbers) after trying several others.

Another possible reason for this finding of fixing errors without no causal model of errors is the use of tinkering—random modifications in code using guess-and-check. Lindi's following report on how she discovered the cause of the error in her team's robot describes tinkering:

> I just kept looking over the commands that I was using, and changing them to see what happened if I changed them and, like, what my robot would do. And so, when I started changing my speeds and my delay, I noticed that my robot would go further, would go shorter, it would turn more, turn less, so that's when I figured out that I had to change those to make it work…. I had mine at like 4.6 or some-

> thing, but if you had it at like 4.7, it'd be too much, so it was … very particular in what it had to be…. I just kept trying numbers until I got to one that works [laughs].

Tinkering has been observed in many studies, especially with novice, less skillful programmers (e.g., Fitzgerald et al. 2008, 2010; Jadud 2005, 2006; Lin et al. 2016; Perkins et al. 1986), including in robotic programming contexts (e.g., Kim et al. 2015; Bers et al. 2014; Silk et al. 2009). Systematic, reflective debugging is needed for preservice teachers to learn what worked or not in code and why and how. Without such learning, their debugging would not improve and neither do that of their future students. Systematic, reflective debugging could be guided through such instructional tools as flowchart (Carver and Risinger 1987), reflective memos (Chmiel and Loui 2004), and pair debugging (McCauley et al. 2008).

Hypothesis generation also seems related to a deduction of a causal model of error. Based on the data of participants who deduced a causal model of error, hypothesis formation was a necessary but not sufficient condition for a causal model of error. That is, without a hypothesis, a causal model of the error was not deduced. In other words, all debugging processes that led to a causal model of the error involved hypothesis formulation although not all debugging processes that included a hypothesis necessarily led to the causal model deduction. This finding seems intuitive since generating a hypothesis is part of debugging process (e.g., Bers et al. 2014), and, without this part, successful debugging cannot be achieved. Modelling to form hypotheses for possible errors has been discussed in the literature highlighting the need for learning to debug (Fitzgerald et al. 2008; McCauley et al. 2008). Considering debugging is often regarded as "an iterated process of developing hypotheses and verifying or refuting them" (Araki et al. 1991, p. 14), without a hypothesis, the debugging process is incomplete, which makes it impossible to understand the cause of error.

Many active examinations seemed to be associated with the causal model deduction but not all active examination led to the causal model deduction. Nonetheless, those who used more than one technique in locating errors ended up deducing a causal model of error. Unlike other participants, Abbie and Emma used mixed error-locating techniques: for the first and second debugging processes, they used both backward reasoning and forward reasoning in serial order, and for the third debugging process, they used forward reasoning in both program order and serial order. In all three debugging processes, they knew what caused the error and how the error was resolved. They also tested multiple hypotheses. They mentioned they were frustrated during the process but they did it anyway to make the robot work as illustrated in Abbie's following comment: "I think we were kind of annoyed … it wasn't a big deal … we really wanted it to be perfect" In contrast to Lindi (discussed in "Most debugging processes involved active examination" section) who was frustrated and asked for the instructor's help, Abbie and Emma made continuous attempts to fix the error despite frustration. These findings call for further investigation of what makes one team's struggle productive but the other's struggle unproductive.

In sum, these findings suggest that effort is needed to ensure teachers end their debugging processes with an understanding of the causal model of error. For example, debugging instruction could include explicit teaching (McCauley et al. 2008). It could also be beneficial to keep a programming journal (Chmiel and Loui 2004) in which preservice teachers analyze, discuss, and reflect on their debugging process to conclude it with a causal model of the error. Explicit debugging instruction to early childhood education teachers may positively influence their future teaching of problem solving (Klahr and Carver 1988).

A few completed debugging not by correcting the error but by eliminating the erroneous portion of the program

A few participants simply eliminated problematic segments of the program rather than fixing them. They simplified their original programming plan to avoid having to fix the error. For example, this comment from Giana indicates that she and her teammate, Maya, received the instructor's help. Still, Giana and Maya could not figure out how to keep the robot from stopping during a series of actions, and they excluded the entire series from their robot programming and lesson design, as described by Giana:

> Well, we didn't end up solving it because it was just taking too long, but we wanted to make it go, figure out how to make it go in a square on its own, make the robot go in a square on its own, and we figured out that actually programming, but then it would stop, and when we pressed the button for it to keep going, it wouldn't go. So we couldn't figure out how to make it go straight, and then we press a button and it would go in a square, and then make it go straight again, so we just deleted the square aspect of the robot just for time's sake.

Similarly, Taylor reported the removal of a problematic portion of code: "I know at one point we were trying to loop, do a loop with something, but … I don't think we ended up keeping it." Removal of a buggy portion of code is a strategy often observed also in text-based programming. While the strategy can be effective in terms of removing the error, it is "a difficult strategy to use effectively" because it can lead to newer related errors and further confuse the programmer (Jadud 2006, p. 76).

A pursuit of the original programming goal or an alternative goal was part of debugging process examined in the study of Bers et al. (2014). In their study, as complexity increased in programming (e.g., use of sensors), more children chose an alternative goal. Some early childhood preservice teachers in the present study seem to have chosen to simplify their programming goals when resolving the error was beyond their capacity. This is corroborated by the fact that the preservice teachers who kept their original programming goal in the present study commented on the ease of error repair. For example, when Olivia was asked if she and her teammate had to alter their programming goal, she responded as follows:

> No. We didn't have to give up and move on. I mean it, programming was actually pretty quick because, when I started to struggle and couldn't remember how we did it, I went back and referred to what we had already done and just tweaked it to fit what we needed it to do. And so, it only took us like 15-20 minutes to do it total.

Sara's comments illustrate that addressing her team's programming error also was not complicated: "we kept the same programming. We just adjusted the numbers we used [in the code]." It is worth noting that the majority of participants kept their programming goal. This may be because their programming goal was aligned with their instructional goals, meaning if they changed their robot programming goal, they would have had to also change their lesson design idea including target grade and content level standards.

It is not surprising that there were preservice teachers who gave up on their original programming goal. In Fitzgerald et al. (2010), this phenomenon was called "work around problem" in which a problematic section of code was replaced with a new section of code without attempting to figure out why the code was problematic. Such a phenomenon is frequently observed among novice programmers (Fitzgerald et al. 2008). Their knowledge

needs to be assessed to see where their knowledge lacks, before and during debugging strategy learning. Moreover, their disposition such as a strong desire for time management and uncertainty avoidance could have led to their error avoidance rather than debugging. Further research is needed in this area to balance foci of programming instruction for future teachers.

Limitations and future research directions

Several limitations were present. First, much of the prior debugging literature referenced in the present study was on text-based programming because debugging during block-based programming has been seldom studied. We believe this study opens the door to applying and modifying text-based debugging strategies in and for block-based programming. Second, the number of participants is small. However, participants' debugging processes varied enough to show differences in addition to commonalities. Third, other data sources could be used to see how debugging processes were related to preservice teachers' programming knowledge (Perkins and Martin 1986) and use of computational thinking concepts (Brennan and Resnick 2012). For example, it could be investigated if and how the use of conditionals is associated with any particular debugging process to examine the relation of a computational thinking concept use to debugging. In this regard, debugging processes could have been analyzed around sequential, selective, and repetitive control structures of code.

Conclusion

General discussion

The investigation of early childhood preservice teachers' debugging processes during block-based programing led to the list of six common errors that they often made and the five themes of findings from their debugging processes. The list and themes are discussed in the "Findings and discussion" section, and only general discussions are included in this section.

Successful programming of early childhood preservice teachers in the present study did not equate to successful learning of programming, including understanding how the code worked. This finding is consistent with findings in previous studies using block-programming in which reusing and remixing of existing code and debugging through tinkering allowed novice programmers to create functioning code (Brennan and Resnick 2012). As argued in Perkins and Martin (1985), these preservice teachers' debugging processes informed their understanding of programming practice and a lack thereof. In turn, such information provided insights into teacher preparation possibilities for their future teaching of young children to program, debug, and problem-solve. While there is some literature listing debugging techniques and teaching methods in text-based programming (Fitzgerald et al. 2010; McCauley et al. 2008; Simon et al. 2008), systematic discussions on these techniques and methods have not been offered in the context of preservice teachers' engagement in block-based programming. Therefore, we discuss a few applicable ones here based on the present study findings.

Finding bugs was indeed difficult for preservice teachers. Active examination was not enough to locate bugs. However, debugging strategies are learnable (Katz and Anderson

1987). Backward reasoning (Katz and Anderson 1987) along with formulating multiple (Simon et al. 2008) high quality hypotheses (Fitzgerald et al. 2008) could lead to success in locating bugs. As discussed earlier, compiled reasoning and its practice using common errors and associated robot behaviors could increase familiarity with code. At the same time, these future teachers should not be given step-by-step job aids. Rather, scaffolding (Lye and Koh 2014; Wood et al. 1976) is necessary to make debugging tasks manageable and yet highlight the critical components of the tasks.

While almost every preservice teacher in the present study mentioned frustration during debugging, many did not give up their programming goals. Rather, they persisted to correct errors, with or without help from others. Nonetheless, the majority did not fully understand how the error was fixed due to no deduction of a causal model. Systematic, reflective debugging could be instrumental to these preservice teachers, using reflection-in-action during debugging and reflection-on-action about debugging performed (Schön 1983) also to prevent tinkering. As introduced earlier, reflective memos (Chmiel and Loui 2004) could be applied to debugging block-based code.

Still, debugging is difficult to novice programmers regardless of how they learn because of its requirement of simultaneous use of multiple new skills (Fitzgerald et al. 2008). It is natural for novice programmers to struggle to complete both comprehension and correction phases in debugging (Yen et al. 2012; Yoon and Garcia 1998). So is their experience of frustration (Fitzgerald et al. 2008). This is where it is logical to think scaffolding is needed for productive struggle. Although Lye and Koh (2014) highlighted scaffolding for frustration control, design for scaffolding to debug needs further research. Literature on productive struggle (Warshauer 2015), productive failure (Kapur 2010, 2011, 2014a, b), desirable difficulties (Bjork and Bjork 2011), and impasse-driven learning (VanLehn et al. 2003) may be of use. Existing debugging games such as BOTS (Liu et al. 2017) and Gidget (Lee et al. 2014) could integrate computer-based scaffolding for novice debuggers' productive struggle.

Implications for research and practice

This study is, to our knowledge, the first systematic study of early childhood preservice teachers' debugging in block-based programming. It provides directions for future research that aims to prepare early childhood teachers for programming, computational thinking, and STEM education. Though this study used robotics as a programming context, findings on debugging processes could be applicable to other contexts involving block-based programming—for example, a game-making context using Scratch. Not only early childhood teachers but also educators in other contexts could benefit from this study. For example, discussions on scaffolding to prepare teachers for scaffolding their future students may inspire computer science teacher educators who are cognizant of the need for teaching not only computer science content but also instructional methods (Yadav et al. 2016) such as scaffolding (Lye and Koh 2014) and pair debugging (McCauley et al. 2008). Possibilities for future research include investigating how teachers debug the code programmed by others (e.g., students) and help them debug. Also, this study demonstrates ways of examining process-oriented design, as opposed to designing something only based on outcomes—successful or unsuccessful programming, for example. Considering debugging is one of the effective ways to cultivate problem-solving skills (Yen et al. 2012), the benefits of this line of research are likely to be beyond computer programming and teacher preparation.

Acknowledgements This research was supported by the National Science Foundation (NSF) under grant 1712286, and internal grants from the University of Georgia (UGA). But any findings, conclusions, or recommendations are those of the author and do not necessarily represent official positions of NSF or UGA.

References

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on innovation and technology in computer science education* (pp. 84–88). New York: ACM. https://doi.org/10.1145/1067445.1067472.

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2007). The impact of improving debugging skill on programming ability. *Innovation in Teaching and Learning in Information and Computer Sciences, 6*(4), 72–87. https://doi.org/10.11120/ital.2007.06040072.

Akcaoglu, M. (2014). Learning problem-solving through making games at the game design and learning summer program. *Educational Technology Research and Development, 62*(5), 583–600. https://doi.org/10.1007/s11423-014-9347-4.

Araki, K., Furukawa, Z., & Cheng, J. (1991). A general framework for debugging. *IEEE Software, 8*(3), 14–20. https://doi.org/10.1109/52.88939.

Belland, B. R. (2014). Scaffolding: Definition, current debates, and future directions. In J. M. Spector, M. D. Merrill, J. Elen, & M. J. Bishop (Eds.), *Handbook of research on educational communications and technology* (pp. 505–518). New York: Springer. http://link.springer.com/chapter/10.1007/978-1-4614-3185-5_39.

Belland, B. R. (2017). *Instructional scaffolding in STEM education: Strategies and efficacy evidence*. http://www.springer.com/us/book/9783319025643.

Bers, M. U. (2010). The tangibleK robotics program: Applied computational thinking for young children. *Early Childhood Research and Practice, 12*(2), n2.

Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers and Education, 72,* 145–157. https://doi.org/10.1016/j.compedu.2013.10.020.

Bers, M. U., Seddighin, S., & Sullivan, A. (2013). Ready for robotics: Bringing together the T and E of STEM in early childhood teacher education. *Journal of Technology and Teacher Education, 21*(3), 355.

Bjork, E. L., & Bjork, R. A. (2011). Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. In M. A. Gernsbacher, R. W. Pew, L. M. Hough, & J. R. Pomerantz (Eds.), *Psychology and the real world: Essays illustrating fundamental contributions to society* (pp. 56–64). New York: Worth Publishers.

Brennan, K., & Resnick, M. (2012). Using artifact-based interviews to study the development of computational thinking in interactive media design. In *Presented at the American Educational Research Association annual meeting*, Vancouver, BC, Canada.

Carver, S. M., & Risinger, S. C. (1987). Improving children's debugging skills. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second workshop* (pp. 147–171). Westport, CT: Ablex Publishing.

Chiu, C.-F., & Huang, H.-Y. (2015). Guided debugging practices of game based programming for novice programmers. *International Journal of Information and Education Technology, 5*(5), 343–347. https://doi.org/10.7763/IJIET.2015.V5.527.

Chmiel, R., & Loui, M. C. (2004). Debugging: From novice to expert. *SIGCSE Bulletin, 36,* 17–21.

Committee for the Workshops on Computational Thinking, and National Research Council. (2010). *Report of a workshop on the scope and nature of computational thinking*. https://doi.org/10.17226/12840.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., et al. (2008). Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education, 18*(2), 93–116. https://doi.org/10.1080/08993400802114508.

Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., & Zander, C. (2010). Debugging from the student perspective. *IEEE Transactions on Education, 53*(3), 390–396. https://doi.org/10.1109/TE.2009.2025266.

Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man–Machine Studies, 7*(2), 151–182. https://doi.org/10.1016/S0020-7373(75)80005-8.

Greiff, S., Wüstenberg, S., Csapó, B., Demetriou, A., Hautamäki, J., Graesser, A. C., et al. (2014). Domain-general problem solving skills and education in the 21st century. *Educational Research Review, 13,* 74–83. https://doi.org/10.1016/j.edurev.2014.10.002.

Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education, 25*(2), 199–237. https://doi.org/10.1080/08993408.2015.1033142.

Gugerty, L., & Olson, G. (1986). Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 171–174). New York: ACM. https://doi.org/10.1145/22627.22367.

Hiebert, J., & Grouws, D. A. (2007). The effects of classroom mathematics teaching on students' learning. In F. K. Lester (Ed.), *Second handbook of research on mathematics teaching and learning* (pp. 371–404). Charlotte, NC: Information Age Publishing.

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education, 15*(1), 25–40. https://doi.org/10.1080/08993400500056530.

Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on computing education research* (pp. 73–84). New York: ACM. https://doi.org/10.1145/1151588.1151600.

Jonassen, D. H. (2000). Toward a design theory of problem solving. *Educational Technology Research and Development, 48*(4), 63–85. https://doi.org/10.1007/BF02300500.

Jonassen, D. H. (2011). *Learning to solve problems: A handbook for designing problem-solving learning environments*. New York: Routledge.

K12 Computer Science Framework Steering Committee. (2016). *K-12 computer science framework*. http://www.k12cs.org.

Kapur, M. (2010). Productive failure in mathematical problem solving. *Instructional Science, 38*(6), 523–550. https://doi.org/10.1007/s11251-009-9093-x.

Kapur, M. (2011). A further study of productive failure in mathematical problem solving: Unpacking the design components. *Instructional Science, 39*(4), 561–579.

Kapur, M. (2014a). Comparing learning from productive failure and vicarious failure. *Journal of the Learning Sciences, 23*(4), 651–677. https://doi.org/10.1080/10508406.2013.819000.

Kapur, M. (2014b). Productive failure in learning math. *Cognitive Science, 38*(5), 1008–1022. https://doi.org/10.1111/cogs.12107.

Katz, I. R., & Anderson, J. R. (1987). Debugging: An analysis of bug-location strategies. *Human–Computer Interaction, 3*(4), 351.

Kazakoff, E., & Bers, M. (2012). Programming in a robotics context in the kindergarten classroom: The impact on sequencing skills. *Journal of Educational Multimedia and Hypermedia, 21*(4), 371–391.

Kim, C., Kim, D., Yuan, J., Hill, R. B., Doshi, P., & Thai, C. N. (2015). Robotics to promote elementary education pre-service teachers' STEM engagement, learning, and teaching. *Computers & Education, 91*, 14–31. https://doi.org/10.1016/j.compedu.2015.08.005.

Kim, C., Yuan, J., Oh, J., Shin, M., & Hill, R. B. (2016). Productive struggle during inquiry learning. *Paper presented at the European Association for Research on Learning & Instruction (EARLI) SIG 20 & SIG 26 Meetings,* Ghent, Belgium.

Kim, C., Yuan, J., Vasconcelos, L., Shin, M., & Hill, R. B. (2017). Prospective elementary teachers' debugging during block-based visual programming. *Paper presented at the American Educational Research Association (AERA) Annual Meeting,* San Antonio, TX, USA.

Klahr, D., & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology, 20*(3), 362–404. https://doi.org/10.1016/0010-0285(88)90004-7.

Lee, M. J., Bahmani, F., Kwan, I., LaFerte, J., Charters, P., Horvath, A.,…Ko, A. J. (2014). Principles of a debugging-first puzzle game for computing education. In *2014 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, Melbourne, Australia (pp. 57–64). https://doi.org/10.1109/VLHCC.2014.6883023.

Lee, M. J., & Ko, A. J. (2015). Comparing the effectiveness of online learning approaches on CS1 learning outcomes. In *Proceedings of the eleventh annual international conference on international computing education research* (pp. 237–246). New York: ACM. https://doi.org/10.1145/2787622.2787709.

Leedy, P. D., & Ormrod, J. E. (2013). *Practical research: Planning and design* (p. c2013). Boston: Pearson.

Lin, Y.-T., Wu, C.-C., Hou, T.-Y., Lin, Y.-C., Yang, F.-Y., & Chang, C.-H. (2016). Tracking students' cognitive processes during program debugging: An eye-movement approach. *IEEE Transactions on Education, 59*(3), 175–186.

Liu, Z., Zhi, R., Hicks, A., & Barnes, T. (2017). Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education*. https://doi.org/10.1080/08993408.2017.1308651.

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior, 41,* 51–61. https://doi.org/10.1016/j.chb.2014.09.012.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., et al. (2008). Debugging: A review of the literature from an educational perspective. *Computer Science Education, 18*(2), 67–92.

Perkins, D. N., Farady, M., Hancock, C., Hobbs, R., Simmons, R., Tuck, T., & Villa, E. (1986). *Nontrivial pursuit: The hidden complexity of elementary LOGO programming*. Reports: Research/Technical No. ETC-TR86-7. Cambridge, MA: Educational Technology Center.

Perkins, D. N., & Martin, F. (1985). *Fragile knowledge and neglected strategies in novice programmers.* IR85-22. http://eric.ed.gov/?id=ED295618.

Perkins, D. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In *Papers presented at the first workshop on empirical studies of programmers* (pp. 213–229). Norwood, NJ: Ablex Publishing Corp. http://dl.acm.org/citation.cfm?id=21842.28896.

Reiser, B. (2004). Scaffolding complex learning: The mechanisms of structuring and problematizing student work. *Journal of the Learning Sciences, 13,* 273–304. https://doi.org/10.1207/s15327809jls1303_2.

Schön, D. A. (1983). *The reflective practitioner: How professionals think in action*. New York: Basic Books.

Schunn, C. D., & Trafton, J. G. (2013). The psychology of uncertainty in scientific data analysis. In G. J. Feist & M. E. Gorman (Eds.), *Handbook of the psychology of science* (pp. 461–483). New York: Springer.

Silk, E. M., Higashi, R., Shoop, R., & Schunn, C. D. (2009). Designing technology activities that teach mathematics. *The Technology Teacher, 69*(4), 21–27.

Simon, B., Bouvier, D., Chen, Tzu.-Yi., Lewandowski, G., McCartney, R., & Sanders, K. (2008). Common sense computing (episode 4): Debugging. *Computer Science Education, 18*(2), 117–133. https://doi.org/10.1080/08993400802114698.

Tsau, S. R. (1996). *College students' diagnostic capabilities in computer programming*. The University of Wisconsin-Madison.

VanLehn, K., Siler, S., Murray, C., Yamauchi, T., & Baggett, W. B. (2003). Why do only some events cause learning during human tutoring? *Cognition and Instruction, 21*(3), 209–249. https://doi.org/10.1207/S1532690XCI2103_01.

Vessey, I. (1985). Expertise in debugging computer systems: A process analysis. *International Journal of Man–Machine Studies, 23*(5), 459–494. https://doi.org/10.1016/S0020-7373(85)80054-7.

Warshauer, H. K. (2015). Strategies to support productive struggle. *Mathematics Teaching in the Middle School, 20*(7), 390–393.

Wood, D., Bruner, J., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry, 17,* 89–100. https://doi.org/10.1111/j.1469-7610.1976.tb00381.x.

Yadav, A., Gretter, S., Hambrusch, S., & Sands, P. (2016). Expanding computer science education in schools: Understanding teacher experiences and challenges. *Computer Science Education, 26*(4), 235–254. https://doi.org/10.1080/08993408.2016.1257418.

Yen, C.-Z., Wu, P.-H., & Lin, C.-F. (2012). Analysis of experts' and novices' thinking process in program debugging. *Communications in Computer and Information Science, 302,* 122–134.

Yoon, B.-D., & Garcia, O. N. (1998). Cognitive activities and support in debugging. In *Proceedings of the fourth annual symposium on human interaction with complex systems* (pp. 160–169). https://doi.org/10.1109/HUICS.1998.659974.