Objects to Debug with: How Young Children Resolve Errors with Tangible Coding Toys

Deborah Silvis, Jody Clarke-Midura, Jessica Shumway
deborah.silvis@usu.edu, jody.clarke@usu.edu, jessica.shumway@usu.edu
Utah State University
Victor R. Lee, Stanford University, vrlee@stanford.edu,

Abstract: Debugging is an important skill all programmers must learn, including preliterate children who are learning to code in early childhood settings. Despite the fact that early learning environments increasingly incorporate coding curricula, we know little about debugging knowledge in early childhood. One reason is that the tangible programming environments designed for young children entail a layer of material complexity that we have yet to account for in terms of learning to debug. In our study of young children learning to program, we found that in the midst of solving programming tasks and learning to debug, tangible toys presented bugs of their own. This paper analyzes video of Kindergarteners learning to debug errors in the program and errors in the physical materials. We argue that concurrent physical and programming bugs present opportunities for young children to learn about the broader computational system in which they are learning to code.

Introduction

Debugging is known to be an important programming skill (Pea, 1986; Pea et al., 1987), and studies of debugging are seeing a resurgence in the learning sciences (DeLiema et al., 2020; Fields et al, 2016; Brady et al., 2020; Kafai et al., 2020). While much of this work deals with the knowledge and computational thinking of novice programmers, there has been less emphasis on how very young children learn to debug. Despite the fact that early learning environments increasingly incorporate coding curricula and CS standards, we know little about programming and debugging knowledge in early childhood (Wang & Choi, 2020). One reason for this gap is that language-dependent frameworks for debugging do not necessarily apply to preliterate programmers. The novel programming environments designed for young children entail a layer of material complexity that we have yet to account for in terms of learning to debug.

Rather than screen-based environments like Logo (Papert, 1980) or block-based tools like Scratch Jr., preschool and Kindergarten children frequently use tangible and hybrid coding toys (e.g. Bers, 2018; Horn, 2018). These types of programming tools include components that can be shared amongst a group of early learners, supporting collaborative debugging activities (Fields et al., 2016). They also introduce a number of new issues for collaborative work, because the materials are distributed and manipulable. Often, in the midst of solving programming tasks and learning to debug, tangible toys present bugs of their own. In this paper, we refer to these types of issues as “physical bugs,” and we examine how children reconcile solving bugs in the domain of the program with concurrent bugs in the physical, material domain.

First we review how physical materials have been treated in studies of debugging, drawing a through-line between Papert’s (1980) floor turtles and today’s tangible toys. Next we briefly describe our design-based study of Kindergarteners learning to code using robot coding toys. We present our analytic process and initial findings that discriminated between buggy problems with the *program* and buggy problems with the *physical materials*. Then we illustrate what resolving these types of bugs looked like when they occurred in isolation and then concurrently, during a series of debugging tasks that a group of Kindergarteners worked to solve in the course of a 30-minute lesson. We show how the objects they used to resolve the coding errors introduced bugs of their own, and we highlight how children debugged both orders of problems. We argue that, rather than constraining or complicating children’s debugging skills, tangible toys present opportunities for children to learn about the larger coding frame or system in which programming occurs.

Framing perspectives: Tangible coding toys and young children’s debugging

Papert’s (1980) floor turtle was one of the first tangible coding toys. Prior to the standard screen-based Logo version, children programmed the turtle to draw basic shapes using a pen that was fitted to its body. Although the floor turtle was more practical for connecting young programmers to the physical world, the robots were expensive, and few made it out of MITs Media Lab and into Kindergartens (Bers & Horn, 2010; McNerney, 2004). No longer tethered to the computer and now increasingly affordable, today’s coding toys bear some resemblance to the first turtles. A growing suite of tangible programming robots can be found in early learning settings (Yu & Roque, 2018).

Describing how children resolved errors in turtle programming, Papert (1980) wrote that "the process of debugging is a normal part of the process of understanding a program" (p. 61). Commonplace as debugging may be among novice programmers, we do not yet have an understanding of what the process of debugging looks like for young children. Some early tangible programming tools that grew out of the Media Lab, such as Perlman's Slot Machine, required children to re-code entire sequences rather than editing buggy programs (McNerney, 2004), a fairly inefficient strategy when the program grows to more than a few commands. In his studies of young children programming in Logo, Pea (1986) referred to a "superbug," the idea that, independent of programming language, novice programmers learn that some meanings need to be expressed explicitly in code, while others do not. Already hardwired, beyond the programming domain at the level of electronics, lies another layer of encoded problems to solve, black-boxed within the robot the children are programming (Resnick et al., 1998; 2000). For young children, debugging using tangible toys involves solving problems at the level of the program and at the level of physical materials (Fields et al., 2016).

Study design: Coding in kindergarten

This study took place in the context of a larger design-based research project called Coding in Kindergarten (NSF #1842116) where we are investigating early childhood computational thinking using commercially available robot coding toys (Clarke-Midura et al., 2021). Participants were 48 children in 3 schools, divided into groups of 3-4 children for coding lessons. Lessons occurred twice per week for one month, and children were assessed following implementations. Coding lessons were designed to elicit knowledge and skills associated with computational thinking (i.e. algorithmic thinking, sequencing, pattern recognition, decomposition, and debugging). During lessons, groups of children and a teacher interacted with a series of tangible robot coding toys and other materials included in the coding kits, such as large floor grids used for planning robot paths. Lessons involved programming a robot to travel on a path through the grid, and "codes" consisted of directional arrows placed in a sequence of movement commands. The tangible programming environment- color-coded directional commands, large floor grids, and multiple manipulatives- supported groups of preliterate children to build and debug programs.

The focal cases in this paper illustrate children's interactions with Botley, a coding robot that operates by entering directional codes into a remote control connected to the agent through Bluetooth (see Figure 1). Botley's kit comes equipped with a set of directional arrow tiles that can be sequenced to build a program. Botley can be used with or without its "arms," which guide a small ball into a goal. We observed that it was sometimes difficult to keep the kit's directional arrow tiles in order when sequencing algorithms, thus we iteratively designed means of organizing the codes, making them easier to manipulate. In the first two cases, children used magnetic code tiles on a metal sheet to build their program. In the third case, they built programs using the code tiles that came equipped with the coding toy kit, along with a paper program organizer designed by members of our research team. One minor detail, important for Botley's operation and for interpreting the cases, is that you must press the "trash" button on the remote control to clear Botley's program before programming and executing a new one.

Data and analytic methods

Approximately 30 hours of classroom coding lessons were video recorded and then content logged (56 individual lessons) (Jordan & Henderson, 1995). We then conducted qualitative coding, which consisted of iterative rounds of open coding for characteristics of bugs and debugging activity followed by axial coding according to an emergent relationship between (1) bugs in the program and (2) problems with the materials (Glaser & Strauss, 1967). Within these two broad categories, we identified types of bugs/problems and strategies for resolving/repairing them, and we conducted frequency counts. Further analyzing patterns in frequency and occurrence of bugs, we found that, rather than discrete bugs solved serially, programming and physical bugs frequently co-occurred. For this paper, we selected two cases that illustrate what it looked like when children only needed to address a programming or a physical bug and contrasted these with a third case of a 30-minute lesson which elicited multiple co-occurrent bugs. We conducted interaction analysis of these debugging episodes, focusing on participants' talk, gesture, and coordinated use of materials (Jordan & Henderson, 1995).

Frequency of bugs and strategies for resolving/repairing

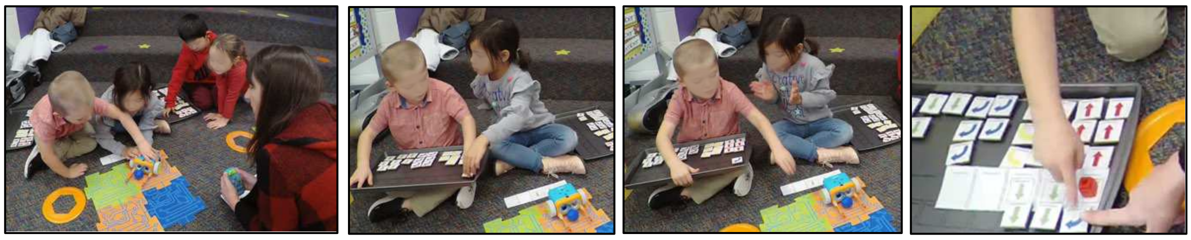
Table 1 summarizes the results of the types and frequencies of bugs and strategies for debugging in both the program and bugs related to physical errors (i.e., physical domain). Overall, bugs in the program most often involved turn errors and missing codes, and errors related to the materials most often involved controller errors. Students' strategies were frequently related to the type of bug. For example, if the students' program was missing a forward code and the robot did not make it to the goal (i.e., *Missing Code*), the debugging strategy was usually to *Add on to End*. In the cases below, while we mention the strategies students used, we focus our analysis on the ways students addressed programming and physical bugs. Case 1 illustrates a situation in which students encountered a *Wrong Start Code* bug (programming bug), and Case 2 provides an example of an *Initializing Robot* bug (physical bug). We use Case 3 to illustrate students' ways of resolving and repairing multiple concurrent bugs.

Table 1: Bug types and strategies for Programming and Physical bugs

PROGRAMMING BUG TYPES: semantic errors when specifying code sequence
Turn Errors (56) *Misunderstanding turn unit (turn-rotate, not turn-turn) or specifying opposite direction*
Wrong Start/End/Middle code (23) *Positional bug errors requiring switch at location in sequence*
Missing Code (63) *Children skip a code or miscount a number of needed rotations or forward moves*
Spurious Code (29) *Inserting or specifying an unneeded, extra code or codes*
Sequencing Error (9) *All codes are present, but out-of-order*
Goal/Path Problem (36) *Losing track of path being sequenced or lack of clarity about endpoint*
DEBUGGING STRATEGIES: process of diagnosing and resolving semantic errors
Swap Codes (46) *Removing a code or codes and replacing them with other codes*
Remove/Add on to End (76) *Editing the end of the program by making the sequence longer or shorter*
Clear Program, Start Over (30) *Removing all codes from the program board and/or pressing delete button*
Remove Code, Move Codes Down (10) *Shortening the program by removing a start or middle code error*
Accommodate Bug (9) *Transform a bug into a feature of the program, change the goal or path*
PHYSICAL BUG TYPES: physical errors in material or mechanical apparatus
Material Mishap (24) *Incidental issue with grids, tiles, or accessories (i.e. robot wheel snags on rug)*
Initializing Robot (38) *Forgetting to reposition robot on start position; false start; mis-orienting robot*
Controller Errors (130) *Remote control, program board (i.e. incomplete button press; forgot to "trash" out)*
Mechanical Issues (35) *Problems with motor, batteries, Bluetooth pairing, or on/off buttons*
Building in a Bug (29) *Intentional user error for pedagogical or personal reasons (i.e. sabotage)*
REPAIR STRATEGIES: process of operating and repairing physical errors
Halt and Re-run Program (56) *Interrupting program execution when a physical bug is identified, restarting*
Work-arounds (12) *Swapping out robot units or materials that are malfunctioning*
Just-in-time Fixes (18) *Catching, mitigating an issue with the physical apparatus before it causes a problem*
Sweeping a Bug under the Rug (35) *Quickly moving on, re-running the program, ignoring physical issue*
Accounting for Technical Issue (12) *Explicitly addressing the physical cause of a failed program*

Case 1: Resolving a programming bug without mechanical interference

Typically, when we think of debugging, we imagine programmers fixing a problem *in the program*. At times during coding lessons, children resolved programming bugs in this commonsense manner, without the materials interfering in debugging. We designed a task called *What Happened?* to explicitly teach debugging strategies for resolving common programming errors. The *What Happened?* task involved giving children buggy code that failed to get the robot to its goal and asking them to debug the program. In one such task, Lacey and Max worked together to diagnose and fix the broken code LEFT FORWARD FORWARD, where the bug was a wrong turn in the start position. The appropriate debugging solution was to swap the start code, replacing LEFT with RIGHT.



[a] Lacey: You have to turn him this way and get him to

[b] Max: No, no, no, the blue

[c] Lacey: That one? Oh, yeah!

[d] Max: We had to change it to blue

Figure 1. Lacey and Max debug a programming bug

Start position bugs involving rotations proved particularly challenging for children; many children applied strategies such as adding on to the end or removing the start code altogether to resolve this bug, resulting in multiple failed debugging attempts. After simulating how they wanted the robot to move, Lacey initially appeared to *re*apply the same wrong turn in their debugged program (Figure 1, a). Max quickly intervened, asserting that they needed the "blue" (i.e. the RIGHT) turn (b). Lacey clapped her hands excitedly to indicate agreement, and they compared their program with the buggy program to specify the location of the bug (c,d). With the robot turned off, Max and Lacey then demonstrated what their debugged code would make Botley do by simulating the movements. They were able to focus on the programming error, since no mechanical issues interfered in their debugging.

Case 2: Reorienting to a physical bug as a potential problem source

Programming bugs are a ubiquitous part of building algorithms, especially for novice programmers, who, in addition to learning a programming language, must also learn language-independent conventions for expressing meanings within a formal system of mechanistic rules (Pea, 1986). One thing all beginning programmers must determine is how much the computer "knows" and how explicit to make their commands in light of the built-in constraints. Tangible coding toys are built to make 90 degree turns, therefore establishing the correct starting orientation before initiating a program is paramount. Regardless of design, all tangible coding robots we use have a "face" that assists children in establishing robot starting positions. Still, forgetting to set the robot's orientation before running a program over multiple debugging attempts was a continuous source of physical bugs. We refer to these bugs as "initialization bugs"; however, rather than written code that would declare and clear variables and set initial states, initialization bugs take place *in the physical, tangible domain* where they represent failures to re-set the robot itself on the proper trajectory.

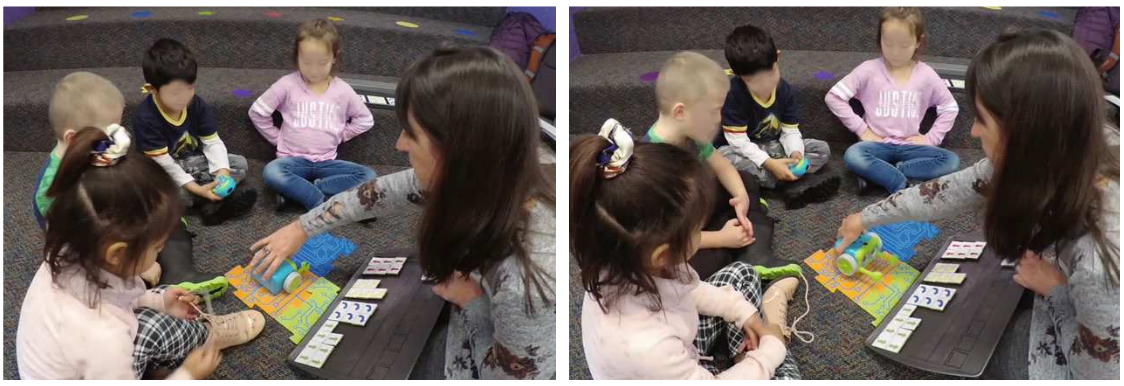
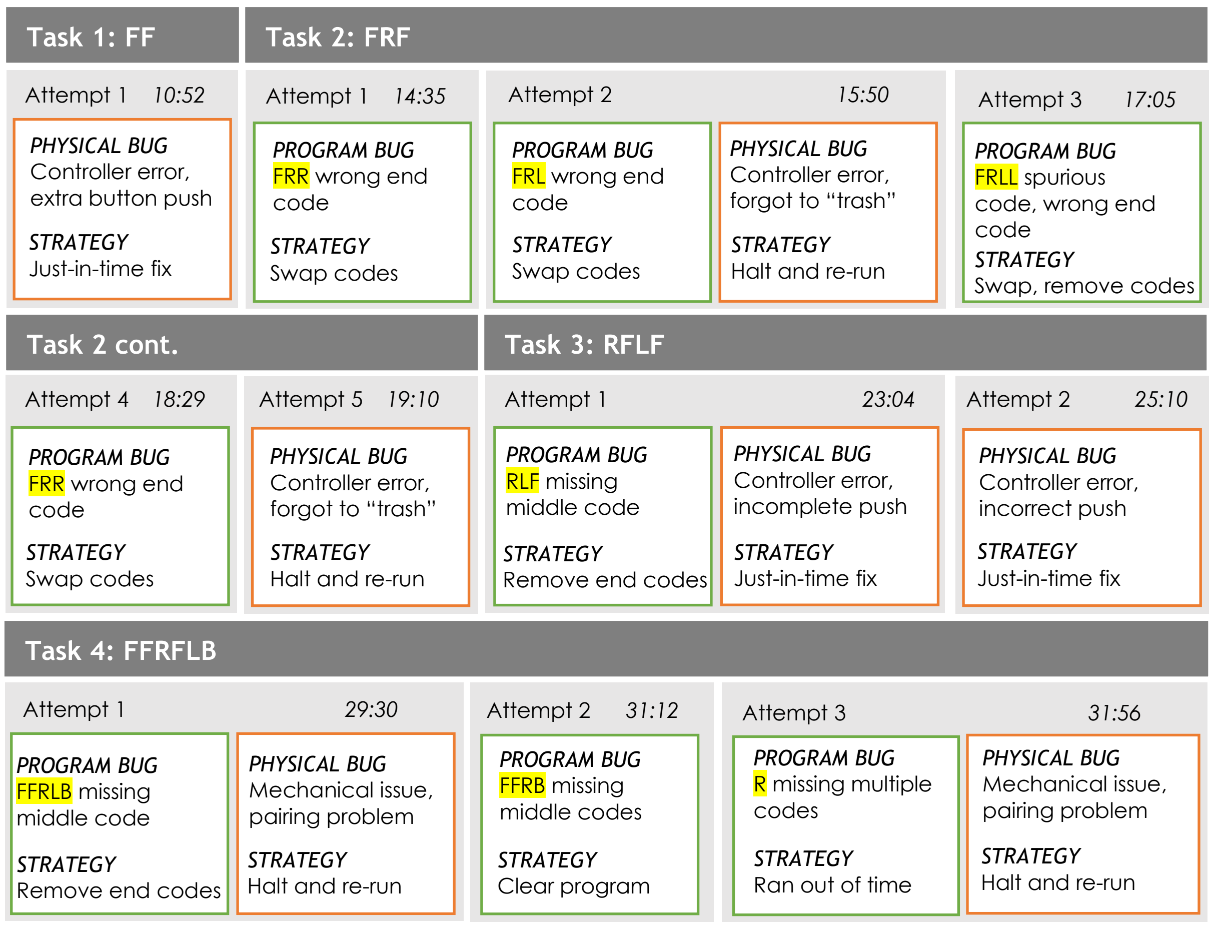


Figure 2. Jessica reorients the robot (left) and then reenacts this strategy (right). *Counter-clockwise from Jessica: Penelope, Paulo, Max, Lacey*

While reviewing Botley's movement-code correspondences during one lesson, the teacher Jessica used a just-in-time re-orientation to probe children's understanding of the critical concept of initializing the robot (Figure 2). The group was about to run a program LEFT FORWARD that was intended to land Botley on the blue tile. However, the robot was mis-oriented from a previous program; in the direction it was facing, LEFT FORWARD would have landed it *off* the tiles and on the carpet in front of Max (a frequent mishap that was a source of endless amusement for children). Just as Paulo pushed the button on the remote to run the program, Jessica realized that the robot needed to be re-oriented and swiftly rotated Botley 90 degrees to the right. After the program successfully reached the blue tile, she then paused to reenact her last-minute rotation, asking the children why they thought she had re-oriented it. Max suggested that Jessica was trying to "control" it (she was) and Lacey offered that it "didn't work" (it did). The children's reasoning about physical problems that require their own sort of debugging was still developing. It was, therefore, important that Jessica prompted them to examine the physical requirements of the programming environment. Doing so at a moment like this, when there was no concurrent problem in the program, supported them to develop repair strategies they would need when programming and physical bugs co-occurred.

Case 3: Reconciling concurrent programming and physical bugs

While the first two episodes show how young children address and resolve bugs, debugging was rarely this simple, and often involved a cascade of recurrent- and concurrent- bugs. Most coding lessons included multiple target programs, each of which required debugging over some number of attempts, which involved resolving both physical and programming bugs. When physical and programming bugs co-occurred, children and teachers needed to find both a debugging strategy that fixed the code *and* a repair strategy that mitigated physical bugs. More often than not, teachers intervened and interrupted physical bugs before they caused the whole program to miscarry, for example, when Jessica applied a just-in-time fix to a potential initialization problem. As children worked together to learn how to program *and* how to use the robots, bugs proliferated, compounding their problem-solving.



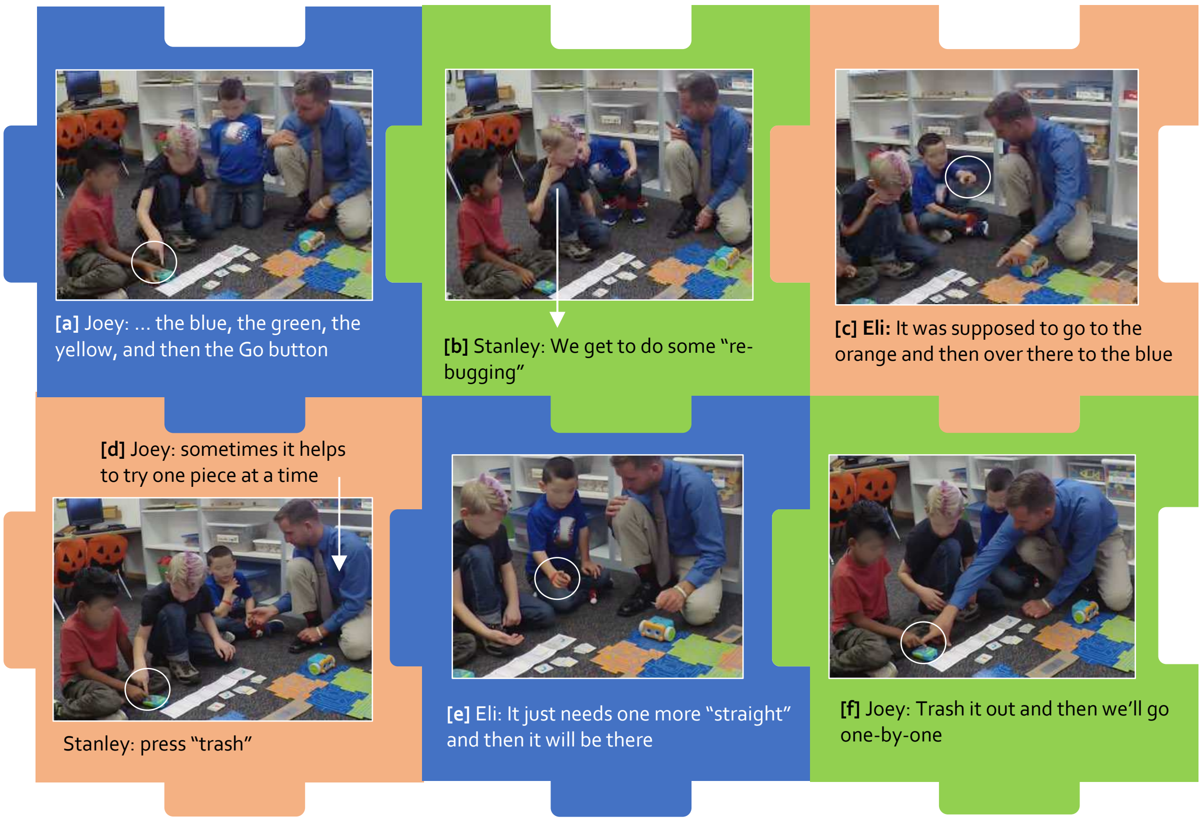
Task	Attempt	Time	Bug	Description	Strategy
Task 1: FF	Attempt 1	*10:52*	*PHYSICAL BUG*	Controller error, extra button push	Just-in-time fix
Task 2: FRF	Attempt 1	*14:35*	*PROGRAM BUG*	FRR wrong end code	Swap codes
Task 2: FRF	Attempt 2	*15:50*	*PROGRAM BUG*	FRL wrong end code	Swap codes
Task 2: FRF	Attempt 2	*15:50*	*PHYSICAL BUG*	Controller error, forgot to "trash"	Halt and re-run
Task 2: FRF	Attempt 3	*17:05*	*PROGRAM BUG*	FRLL spurious code, wrong end code	Swap, remove codes
Task 2 cont.	Attempt 4	*18:29*	*PROGRAM BUG*	FRR wrong end code	Swap codes
Task 2 cont.	Attempt 5	*19:10*	*PHYSICAL BUG*	Controller error, forgot to "trash"	Halt and re-run
Task 3: RFLF	Attempt 1	*23:04*	*PROGRAM BUG*	RLF missing middle code	Remove end codes
Task 3: RFLF	Attempt 1	*23:04*	*PHYSICAL BUG*	Controller error, incomplete push	Just-in-time fix
Task 3: RFLF	Attempt 2	*25:10*	*PHYSICAL BUG*	Controller error, incorrect push	Just-in-time fix
Task 4: FFRFLB	Attempt 1	*29:30*	*PROGRAM BUG*	FFRLB missing middle code	Remove end codes
Task 4: FFRFLB	Attempt 1	*29:30*	*PHYSICAL BUG*	Mechanical issue, pairing problem	Halt and re-run
Task 4: FFRFLB	Attempt 2	*31:12*	*PROGRAM BUG*	FFRB missing middle codes	Clear program
Task 4: FFRFLB	Attempt 3	*31:56*	*PROGRAM BUG*	R missing multiple codes	Ran out of time
Task 4: FFRFLB	Attempt 3	*31:56*	*PHYSICAL BUG*	Mechanical issue, pairing problem	Halt and re-run

Figure 3. Outline of debugging during a 30-minute coding lesson.

Next, we sketch the anatomy of a coding lesson (comprised of a series of tasks, target programs, attempts, concurrent bugs, and debugging discussions) (Figure 3). We then zoom in on a single task in the lesson to show how the group organized their shared work and materials to handle programming and physical errors. This particular lesson involved an activity design called *Crack-the-code*, in which children attempted to program one Botley to run the hidden code of another Botley after they watched its movements. The children alternated between planning the program, sequencing the arrows, and pushing the commands on the remote control, sharing the materials and rotating roles between tasks. After solving a number of concurrent programming and physical bugs in the previous two tasks and cracking the code for the programs FORWARD FORWARD (FF) and FORWARD RIGHT FORWARD (FRF), the children attempted to crack a third target program: RIGHT FORWARD LEFT FORWARD (RFLF). After watching the first Botley run the program several times, Eli attempted to replicate it, instructing his programming assistant Stanley to sequence the codes RIGHT LEFT FORWARD (RLF).

At this point, they had produced a single programming bug; the program was missing a middle FORWARD code, for which the most efficient debugging strategy was to move the two end codes down and insert the missing code. However, before they ran the buggy program to test it, they first had to avoid potential user errors. After Joey reminded Christian to "trash out" the remote to delete the previous program, Christian began to enter the three codes into the remote, but Joey noticed that Christian pushed one of the buttons only half-way (the robot lights up when a full button push is registered). Joey stopped Christian inputting commands and instructed him to "push it harder." Joey called out the colors that corresponded with the arrows, and Stanley assisted Christian by pointing to the buttons (Figure 4, a). Joey's and Stanley's assistance with the physical material prevented user errors that would potentially have muddled Eli's programming bug.

After they ran this first buggy program and the robot failed to land where they intended it to, Stanley exclaimed excitedly that they get to do some "re-bugging" (b), coining a term that aptly describes children's recurrent attempts to debug with robot coding toys. When Joey asked Eli if there was anything he wanted to "fix" or "change," Eli was initially unsure. He began to demonstrate over the robot which way he wanted it to turn, referencing directions and colors of the tiles (c). As part of the process of learning to debug, children often described what they wanted the robot to do by referencing the physical grid space before blending these movements with the relatively unfamiliar and abstract domain of the program (Silvis et al., 2020). Eventually, Joey suggested that they remove the end codes and test each code one-by-one (d). This was a basic strategy teachers used with novice coders, who were more successful debugging programs when they could simply incrementally add onto the end of the sequence, fixing codes one at a time as necessary.


[a] Joey: ... the blue, the green, the yellow, and then the Go button
[b] Stanley: We get to do some "re-bugging"
[c] Eli: It was supposed to go to the orange and then over there to the blue
[d] Joey: sometimes it helps to try one piece at a time
Stanley: press "trash"
[e] Eli: It just needs one more "straight" and then it will be there
[f] Joey: Trash it out and then we'll go one-by-one
Figure 4. Debugging during Task 3

Following instruction from Stanley about trashing out the program in the remote control and pushing the Go button, Christian ran a single RIGHT command. Having isolated the initial movement, Eli then sequenced the remainder of the new program with Stanley's help (e). It was again time for Christian to program the codes into the remote control. Having observed multiple remote control-related user errors, the whole group was by then carefully monitoring Christian's operation of the remote. He successfully pushed the blue (RIGHT) and green (FORWARD) buttons, *but then mistakenly pushed another forward button* before the right turn button. Joey told him to press trash and start over, a just-in-time repair strategy in the physical domain that circumvented program failure. Joey pointed to the trash button and then told Christian to push the buttons "one-by-one" (f). Stanley pointed to each of the arrows in the program as they called out the arrow colors to Christian. In the course of the RFLF task, the children had applied both debugging strategies to the program and repair strategies to the physical materials. One-by-one coding and one-by-one button pushing were required to resolve errors in the program *and* prevent errors in the physical materials. We turn in the discussion to the ways in which tangible programming involves learning to debug both domains.

Discussion: Objects to debug with

In this paper, we presented the idea that tangible programming lends itself to forms of debugging at the level of the program and in the domain of the physical materials. Because coding robot toys are comprised of multiple, manipulable components, bugs occurring in the physical domain, such as forgetting to delete the previous program or pressing the wrong button on the remote control, can make it challenging for children to tease out errors in the program. Teachers' just-in-time strategies for addressing physical bugs- and for making technical requirements more transparent for children- may partly mitigate confusion; however, children must still grapple with the relationship between errors in their programs and errors due to the materials. As such, tangible toys can be thought of as "objects to debug with," which introduce novel orders of problems to solve. In a programming context where the success of a program hinges as much on accurately sequencing as it does on reliably inputting codes, operating materials and tangible programming go hand in hand.

Another characteristic of this particular computational environment that is consequential for coding involves the semantics of the coding language. The operative symbol system is a collection of color-coded arrows, a language that instantiates the meaning of directional commands in a symbolic form legible to preliterate children. However, Kindergarten children are also developing early spatial reasoning skills that implicate these same directional symbols; thus, learning the movement conventions of tangible toys (i.e. how the robots operate in 3D space) is entangled with learning the movement-symbol correspondence (i.e. how the symbols operate within a program) (Silvis et al., 2020). This complexity is compounded when the physical materials malfunction, or when there are user errors. The children in the cases we presented above are simultaneously learning *at least* the following concepts and conventions: (1) to perform mental rotation in 3D space (2) that the turn commands do not move the robot to another square, but merely rotate them on the same square (3) that the sequential actions of the robot depend on a precise sequence of codes and (4) how to input this same planned sequence of codes into the remote control.

Given the constraints built into the robots and this series of correspondences children are learning, concurrent physical and programming bugs require children to grapple with the larger programming environment or computational system. Whereas coding toys may appear straightforward and "user friendly" compared with screen-based or hybrid alternatives, tangible programming often involves solving both semantic (i.e. programming) and pragmatic (i.e. environmental) problems. Rather than a design flaw of tangible toys, we see this complexity as a resource for learning to debug, because it forces children to consider the broader relationship between all the components of the computational system: the codes; their meanings; their function (i.e. to give instructions to the robot); the robot and its movement constraints; coding robot accessories like the remote control (how does it "talk" to the robot, anyway?). Learning to coordinate these computational components to solve problems is a tall order in Kindergarten- especially when groups of children program collaboratively- but one that we see as productive for learning and consequential for their engagement with computing down the road.

Finally, while we focused in this analysis on the relationship between concurrent programming and physical bugs and the consequences of these co-occurring bug *types*, we also see a need for more study of the corresponding *strategies* students and teachers used to resolve bugs in both domains. Successfully resolving bugs requires an alignment between bug type and solution strategy. When children attempt to debug a programming bug in the middle of the program by adding onto or removing codes from the end, they fail to debug the program. Teaching children to apply particular debugging strategies to particular types of bugs is important across programming environments, no less in tangible programming (Klahr & Carver, 1988; McCauley et al., 2008). However, coding robot toys also require learning to apply a strategy to fixing the programming bug that teases out and takes into account the presence of concurrent physical bugs. This means placing pedagogical focus on the technical requirements/constraints of toys and teaching children not only how to fix problems with their programs but also how to diagnose and repair problems with operating the robots. Tangible programming environments expose how learning to debug does not simply follow from learning to program. We need to design for debugging (e.g. Fields et al., 2016), and treat tangible coding toys as objects to debug with.

Conclusion

The recognition that debugging is an important skill for programming has led to a number of studies. However, most of these studies focus on older students who are literate and interacting with computer screens. Very few studies focus on how young, preliterate children learn to debug. Integrating CS into early childhood requires a deeper understanding of the context and materials with which young children engage while learning to program. Preschool and kindergarten aged children often use tangible and hybrid coding toys that use directional arrows as a syntax. Programming these robot toys requires children to push buttons or manipulate tiles, often external to the toy. Thus, understanding how young children debug requires understanding the level of complexity the materials introduce and the nature of both the programming and physical bugs they encounter. Insofar as young children are learning how to operate these materials while they are learning to code, interactions with the toys are consequential for learning. More study is needed of the pedagogical approaches that support children to align their debugging strategies with the types of bugs they encounter. As learning designers, attending to tangible programming toys as objects to debug with provides insight into young children's thinking and supports them developing valued computational skills and knowledge.

References

Bers, M.U. (2018). *Coding as playground: Programming and computational thinking in the early childhood classroom*. Routledge.

Bers, M. & Horn, M.S. (2010). Tangible programming in early childhood: Revisiting developmental assumptions through new technologies. In I.R. Berson & M.J. Berson (Eds.), *Childhood in a Digital World*. Information Age Publishing.

Brady, C, Gresalfi, M., Steinberg, S., Knowe, M. (2020). Debugging for art's sake: Beginning programmers' debugging activity in an expressive coding context. ICLS 2020.

Clarke-Midura, J., Silvis, D., Lee, V., Shumway, J., Kozlowski, J. (2021). Developing a Kindergarten computational thinking assessment using evidence-centered design: The case of algorithmic thinking. *Computer Science Education*, doi.org/10.1080/08993408.2021.1877988.

DeLiema, D., Dahn, M., Flood, V.J., Abrahamson, D, Enyedy, N. & Steen, F. (2020). Debugging as a context for collaborative reflection on problem-solving processes. In E. Manolo (Ed.), *Deeper Learning, Dialogic Learning, and Critical Thinking: Research-Based Strategies for the Classroom*, p. 209-228. Routledge.

Fields, D., Searle, K.A., & Kafai, Y.B. (2016). Deconstruction kits for learning: Students' collaborative debugging of electronic textile designs. In P. Blikstein, M. Berland, & D.A. Fields (Eds.), *Proceedings of the 6th Annual Conference on Creativity and Fabrication in Education (pp.82-85). New York, NY: ACM.*

Glaser, B.G. & Strauss, A.L. (1967). *The discovery of grounded theory: Strategies for qualitative research.* Aldine Publishing.

Horn, M. (2018). Tangible interaction and cultural forms: Supporting learning in informal environments. *Journal of the Learning Sciences, 27*:4, 632-665.

Jordan, B. & Henderson, A. (1995). Interaction analysis: Foundations and practice. *The Journal of the Learning Sciences, 4*(1), 39-103.

Kafai, Y. et al. (2020). Turning bugs into learning opportunities: Understanding debugging processes, perspectives and pedagogies. ICLS 2020.

Klahr, D. & Carver, S.M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology, 20*, 362-404.

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

McNerney, T.S. (2004). From turtles to tangible programming bricks: Explorations in physical language design. *Personal Ubiquitous Computing, 8*: 326-337.

Papert, S. (1980). *Mindstorms*. Basic Books.

Pea, R.D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research, 2*(1), pp. 25-36.

Pea, R.D., Soloway, E., Spohrer, J.C. (1987). The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics, 9*(1), pp. 5-30.

Resnick, M., Berg, R. & Eisenberg, M. (2000). Beyond black boxes: Bringing transparency and aesthetics back to scientific investigation. *Journal of the Learning Sciences, 9*:1, 7-30.

Silvis, D., Lee, V., Clarke-Midura, J., Shumway, J., & Kozlowski, J. (2020). Blending everyday movement and representational infrastructure: An interaction analysis of Kindergarteners coding robot routes. In M. Gresalfi & L. Horn (Eds.), *Proceedings of International Conference of the Learning Sciences (ICLS).*

Wang, X.C. & Choi, Y. (2020). Teacher's role in fostering preschoolers computational thinking: An exploratory case study. *Early Education and Development*. https://doi.org/10.1080/10409289.2020.1759012.

Yu, J. & Roque, R. (2019). A review of computational toys and kits for young children. *International Journal of Child-computer Interaction, 21*, 17-26.