Assessing Computational Thinking through the Lenses of Functionality and Computational Fluency

Shari J. Metcalf, Joseph M. Reilly, Soobin Jeon, Annie Wang, Allyson Pyers, Karen Brennan and Chris Dede

Harvard Graduate School of Education, Harvard University, Cambridge, USA

Shari J. Metcalf, Joseph M. Reilly, Soobin Jeon, Annie Wang, Allyson Pyers, Karen Brennan & Chris Dede (2021) Assessing computational thinking through the lenses of functionality and computational fluency, Computer Science Education, DOI: 10.1080/08993408.2020.1866932

Assessing Computational Thinking through the Lenses of Functionality and Computational Fluency

Background and Context: This study looks at computational thinking (CT) assessment of programming artifacts within the context of CT integrated with science education through computational modeling. Objective: The goal is to explore methodologies for assessment of student-constructed computational models through two lenses: functionality and conceptual fluency. Method: This study uses data from research with EcoMOD, a 3rd grade ecosystem science curriculum in which student pairs program computational models of a beaver building a dam. Snapshots of programs for 47 student pairs collected over time are assessed. Findings: A functionality-based rubric provided assessment of student task progress, but was less successful at correctly identifying CT gains in programs that were partially correct. A rubric for conceptual fluency identified development of fluency in CT concepts of sequencing, loops, and conditionals. Implications: This study contributes to the literature by exploring affordances of different rubric-based CT assessments of student programs.

Keywords: Computational Thinking; Assessment; Block-based Programming

Introduction

The importance of computational thinking (CT) (Wing, 2006) as a learning goal has been increasingly recognized by researchers and educators (Lye & Koh, 2014; Grover & Pea, 2018; Israel et al., 2015). CT can be generally defined as a way of thinking that includes many of the skills and practices that are part of computer programming, such as algorithmic thinking, logic, abstraction, generalization, decomposition, and debugging, but are also applicable to many areas beyond programming. (Buitrago, Flórez et al., 2017).

CT is recognized as a foundational competency for problem solving in STEM contexts (Grover & Pea, 2018; NGSS Lead States, 2013) Approaches to integrating computational thinking and STEM education include its use for computational modeling – using computational tools to develop models of scientific phenomena, and

promoting learning through the creation, testing, and manipulation of these models (Sengupta et al. 2013; Weintrop et al., 2016; Aksit & Wiebe, 2020; Wagh et al, 2017). Modeling is a fundamental practice of science (Giere, 1988; Nersessian, 2008; Lehrer, 2009), and CT aligns strongly with science learning as a means of representing scientific phenomena through the construction of computational models (Sengupta et al., 2013).

Assessment of CT is a growing field of research, with many forms of CT assessment developed and tested with students at different grade levels, and based on different CT definitions, models, and theoretical frameworks (Tang et al., 2020). A wide range of methodologies are used for CT assessment, including multiple-choice or open-ended tests, performance or task-based assessments, and portfolio assessments (Adams et al., 2019). Some assessments focus on CT concepts, while others include practices as well (Shute et al., 2017). The goal of embedding CT in STEM contexts brings challenges about the assessment of CT as it is integrated in different subject areas, and how those varied contexts inform the ways in which CT can be assessed (e.g., Hutchins et al, 2019).

This paper discusses methodologies to assess students' computational thinking through examining their programmed computational models. For this research, we have chosen to develop assessments of student performance on a specific task involving computational modelling as part of a 3rd grade science curriculum. Using a curriculum that involves students in developing computational models, we present two complementary methodologies for assessing the code of these programs. Each methodology acts as a distinct lens on the data and defines the understandings we are able to measure; the strategies used to analyze the data provide different information for assessing student learning. We share our findings about the aspects of computational

thinking that can be assessed using each type of analysis and discuss the implications of each rubric for student assessment. The information about student errors and hurdles that emerged from our analysis was also used to inform improvements in the curriculum and teacher professional development materials that formed the basis of our study.

The data for this study are derived from an elementary curriculum called EcoMOD (Dickes et al., 2019) in which students are engaged in programming computational models of scientific phenomena. EcoMOD is an ecosystems science curriculum that blends an immersive virtual environment with an agent-based computational modeling tool to support growth in computational thinking and ecosystems science for 3rd grade learners (age 8 to 9 years old). During the 14-day curriculum unit, student pairs spent two class periods programming models of a beaver building a dam. Logfile data collected on pairs' programming activities included snapshots of student code whenever they ran their programs. For this study, we analyzed these program snapshots to assess developing fluency in computational thinking concepts.

The rest of the paper is organized as follows. We first present related work in CT, computational modeling, block-based programming, and CT assessment. We describe the goals of the study, and our methods, including the design of the EcoMOD curriculum and programming tools, the study design, the classroom research that was conducted, and the two types of methodological analyses we performed with the study data. We present our findings for each of the two types of analysis, and conclude with a discussion of the findings and insights on the design of CT assessments.

Literature Review

Computational Thinking and Programming

Programming and CT are deeply intertwined; most CT research involves students learning programming as a means to learn the concepts and skills related to computational thinking (Lye & Koh, 2014; Israel et al., 2015). Brennan and Resnick (2012) characterize programming as a "setting for developing capacities for computational thinking," and Shute, Sun, and Asbell-Clarke note that, while CT and programming are not the same, "being able to program is one benefit of being able to think computationally" (2017, p. 5).

Brennan and Resnick (2012) developed a CT framework in the context of the visual block-based programming language Scratch. It involves three key dimensions: 'computational concepts' (sequences, loops, events, parallelism, conditionals, operators, and data); 'computational practices' (experimenting and iterating, testing and debugging, reusing and remixing, abstracting and modularizing); and 'computational perspectives' (expressing, connecting, and questioning). Rich et al. (2017) derive learning trajectories for K-8 computational thinking from the research literature, describing trajectories for three of those computational concepts: sequence, repetition (or looping), and conditionals. The K-8 learning goals for these three concepts can be summarized as (Brennan & Resnick, 2012; Rich et al., 2017):

- Sequence: A task is expressed as a sequence of actions. The order in which the actions are carried out can affect the outcome.
- Looping: Many tasks include repeated actions. Loops are a mechanism for repeating an action or sequence of actions multiple times.
- Conditionals: Different conditional states may cause different actions to be carried out. More advanced learning includes the use of multiple or

overlapping conditionals, and the use of conditionals with loops, e.g., to determine when to stop a repetition.

For our research, we focus on student learning of these three core computational concepts: sequences, loops, and conditionals, during the EcoMOD programming activity.

Computational Modeling for Science Learning

Recently, there has been substantial work that looks at integrating CT with K-12 science learning (Sengupta et al., 2013; Wagh et al., 2017; Aksit & Wiebe, 2020; Lee, Martin & Apone 2014; Weintrop, et al., 2016; Ryu, Han, & Paik, 2015), primarily in the form of *computational modeling*. Computational modeling can be considered a subset of computational thinking as applied to STEM education (Sengupta et al., 2013; Weintrop et al., 2016). And likewise, it has long been recognized that a benefit of constructing scientific models is this provides a meaningful way to learn programming (Papert, 1991).

Studying a phenomenon by constructing a model provides a means for students to organize and test their knowledge of science concepts by converting concepts and relationships into computational structures that can be executed to generate model behaviors (Sengupta et al 2013). Decomposing a phenomenon into the steps needed to construct a model can make concepts and relationships more explicit (Hutchins, 2020), and the real-time animations and graphs provided by computational models offer scaffolding for interpreting and understanding the modeled phenomena, as well as a means for model validation (Sengupta et al., 2013; Yoon et al., 2016).

Block-based Programming and Modeling

Visual programming tools and environments have made programming much more

accessible to learners, and at an earlier age (Weintrop et al., 2018). Block-based programming languages such Scratch (Resnick et al., 2009) have been a significant breakthrough for learning programming. Block-based languages are important for novice learners because programming is done by dragging and dropping program blocks together, a feature which eliminates the chance for syntax errors (Weintrop & Wilensky, 2017). Block-based interfaces engage young students in intuitive and engaging ways with code, including a palette of blocks that display the available set of commands, as well as visual cues (e.g., color, shape) that indicate block categories and usage. Further, many block-based programming environments are interpreted, rather than compiled, a feature that promotes tinkering (Maloney et al., 2010). Users can cycle between editing and running code with immediate feedback, able to quickly move blocks in and out of a workspace as they run and debug projects.

Similarly, the accessibility of block-based languages has made it possible to develop computational modeling tools for STEM learning in younger grades (e.g., Dwyer et al., 2013; Wagh et al, 2017; Aksit & Wiebe, 2000).

Agent-based modeling is a type of modeling found to be particularly effective for science learning (Dickes et al., 2016; Klopfer et al., 2005; Weintrop et al., 2016). An "agent" is a computational object that is programmed with simple rules. As the agent interacts with its environment and/or other agents, in a 2D graphical visualization, emergent, observable effects appear over time. Again, visual block-based authoring environments have been found to make agent-based programming more accessible. For example, ViMap (Sengupta et al., 2015) provides an abstraction layer over NetLogo, allowing students to focus on their modeling tasks without being overwhelmed by the NetLogo programming language syntax (Hutchins et al., 2020). Sengupta, Kinnebrew, Basu, Biswas, and Clark (2013) describe design strategies to make agent-based

modeling accessible to elementary learners using a visual programming interface, including using the right level of programming primitives, providing means to step through and visualize code execution as an aid to debugging, and making it easy to iterate through build-test cycles.

Computational Thinking Assessment

Numerous studies have examined strategies for the assessment of CT; overviews of the state of the field are presented in recent publications (Tang, et al., 2020; Adams et al., 2019). The most common assessment strategies involve multiple-choice or open-ended tests. For example, the Computational Thinking Test (CTt) for middle school learners includes 28 items and assesses seven CT programming concepts synthesized from CT frameworks, including sequences, loops, and conditionals (Roman-Gonzalez, Pérez-González, & Jiménez-Fernández, 2017). Some assessment tests try to minimize familiarity with specific computing platforms, such as an assessment by Chen et al., (2017) that includes pre-post items measuring both coding in robotics and reasoning in everyday events. Others, like Grover and Basu (2017), use assessment items in a specific programming language (Scratch) to measure understanding of programming constructs.

Another strategy for assessment uses performance or portfolio assessment. In some cases, researchers create tasks for students to complete and use a rubric to evaluate their work products (Werner, et al., 2012; Franklin et al., 2017). In others, students are given an assessment that is task-based, such as the Bebras tasks (Dagiene & Stupuriene, 2016), designed for an international computer science contest, but also used for CT assessment (Román-González, Moreno-León, & Robles, 2017). Sometimes, the program or portfolio produced by students is created during open-ended design

activities, which are then analyzed for evidence of CT understanding (Brennan & Resnick, 2012).

When student artifacts are analyzed for understanding of CT, analysis can include a determination of what level of understanding students have achieved on specific concepts or practices. For example, the application Dr. Scratch (Moreno-León et al., 2015) performs automated portfolio analysis of Scratch projects. To assign a CT score, Dr. Scratch infers competence on seven CT concepts based on analysis of code blocks; for example, competence of flow control is scored as *basic* if the user uses sequences of blocks, *developing* for use of "repeat" and "forever", and *proficient* for use of "repeat until" blocks. Franklin et al. (2017) describe analysis of student programs through identifying milestones for student progress in demonstrating understanding of CT concepts, the milestones representing different levels of proficiency. Artifact analysis can also identify particular challenges or hurdles for students; for example, Blikstein et al. (2014) used learning analytics to identify "sink-states" where students got stuck in coding.

Some evaluations of programming artifacts also look at intermediate steps or iterations of the program as it is being composed. The advantage of looking at student programming in process is this can be a means to see students' progression while solving a problem, and to gather patterns about student learning progressions and challenges (Villamor, 2000; Blikstein et al., 2014; Worsley & Blikstein 2013; Lane & VanLehn, 2005). For example, Troiano et al. (2019) used Dr. Scratch on snapshots of student programs captured over time, rather than just their final products, in order to analyze trajectories in student CT development.

Goals of this study

As part of our research with EcoMOD, we designed a custom block-based programming

tool for agent-based programming, with which students construct agent-based models of a beaver building a dam. In this paper, we describe our assessment of the agent-based modeling programs constructed by student pairs during a study conducted using the EcoMOD curriculum. The assessment process was design-based, and we examined student artifacts through two lenses. First, we conducted an analysis based on their final programming product, using a rubric-based assessment to determine the success of their code in executing the steps for the beaver to build a dam; we considered this a measure of functionality. We found a mismatch between the scoring and our intuition as educators, particularly for assessment of partially successful models, in evaluating the pairs' level of CT understanding. We therefore developed a second analytic approach that was more faithful to student learning, using a rubric-based assessment to capture developing fluency in the use of CT constructs of sequences, loops, and conditionals. We considered this a measure of conceptual fluency complementary to our measure of functionality. Through both of these lenses, we also identified common errors and hurdles that informed the design of our student-facing and teacher professional development materials.

Research Questions

Our research questions for the study are as follows:

(1) What aspects of computational thinking in EcoMOD, if any, can be assessed via a functionality rubric?
(2) What aspects of computational thinking in EcoMOD, if any, can be assessed via a conceptual fluency rubric?
(3) How can CT assessment of program artifacts inform EcoMOD curriculum design?

Methods

EcoMOD Curriculum

EcoMOD is a 14-day, inquiry-based curriculum for 3rd grade that focuses on ecosystem interactions, science practices around modeling, and computational thinking. In EcoMOD, students explore a forest ecosystem and learn about behaviors and causal interactions related to a beaver building a dam. When the beaver builds a dam, it creates a pond, with cascading effects over time on the landscape and other species.

EcoMOD is centered on two computational tools: a 3D immersive virtual world and a 2D modeling tool. In the 3D world (Figure 1, left), students engage with a realistic forest ecosystem, to explore, observe, collect data, and travel in time. The 3D virtual world also includes a point-of-view (POV) tool that gives students the opportunity to "be" a beaver building a dam. The experience of being a beaver helps students learn the steps involved in building a dam, as well as avoiding the predatory wolf.

Figure 1: 3D immersive virtual ecosystem (left) and Beaver POV tool (right).

The 2D modeling tool supports computational modeling activities, as students program an agent-based model of a beaver building a dam (Figure 2, left), using domain-specific visual programming blocks. Model outcomes are visible in a 2D sandbox that represents an abstracted version of the ecosystem (Figure 2, right). The sandbox includes other species, such as mallards and trout. These agents are not programmable but do respond to changes in the ecosystem. There are controls for

different setup options (e.g., the width of the stream, the density of the trees, whether a wolf is present). Graphs allow students to view the emergent outcomes of their models.

Figure 2: 2D model of a beaver building a dam (left) Data graphs in 2D tool (right)

There are four types of blocks (Figure 3): *action blocks* such as "move toward" and "bite tree", *action variables* used to specify an object (e.g., tree, log), *conditional blocks* ("if," "if/otherwise," and "repeat until"), and *conditional variables*, i.e., Boolean variables, to test state (e.g., "at tree," "have log"). The workspace has an implicit forever loop: when the program is running, the program steps through the code blocks in the workspace in a continuous loop. For example, if the "move toward tree" block is placed in the workspace and the play button is clicked, the beaver will take one step toward the closest tree, and then loop, taking another step, then another, until the beaver is at the tree and stays there.

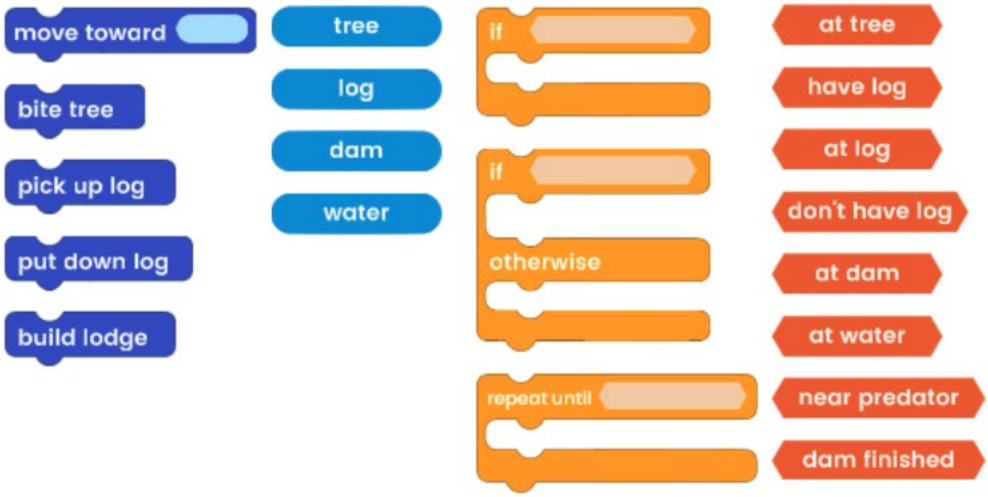


Figure 3: EcoMOD programming blocks

A message window at the bottom of the screen displays feedback messages for assistance in debugging. For example, if the “bite tree” block is executed when the beaver is not at a tree, a message from the beaver says, “I am not at a tree so I can’t bite a tree.” Some errors prevent the program from running, for example, a conditional block without a variable filled in, while others (like the “bite tree” example) just skip the block’s action.

Within this constrained task and limited set of blocks, there remains a range of possible working solutions. Three examples of working models developed by student pairs in this study are provided in Figure 4. In the three examples, the action blocks follow roughly the same sequence, but vary in choices of conditionals and use of nested conditionals.

Figure 4: Three successful computational models for the beaver building a dam.

Beyond a very limited introduction to the 2D modeling tool, no formal instruction in programming is included in the curriculum. The teacher explains the four types of blocks, and demonstrates the interface: how the workspace works, how to drag code blocks in and out, run the program, stop, reset, and change the speed. Students use the 2D modeling tool on days 4 to 7 of the 14-day EcoMOD curriculum. During the first two programming days, they work on programming a model of beaver building a dam, following the steps they observed and enacted in the 3D world. On the beginning of the third programming day, the teacher brings the students together to build a functional "class model" based on suggestions from students and discussion about the programming. Students who do not have a working model are encouraged to copy the class model at this point, so that they can transition to using their now functional models to explore effects of the beaver dam on the rest of the ecosystem.

Participants

This study examined data from five classes of 3rd graders (ages 8-9 years old) from three schools in three different school districts in New England, led by four different teachers. Permission slips were sent home with parents to approve participation in the study. For this study, students worked in pairs on a shared computer, with occasional exceptions when there were an odd number of students in the class or if there was a student absence. In these cases, unpaired students worked individually. After filtering for permission, data from 47 pairs were analyzed for this study. Teachers encouraged students to take turns when programming. As they engaged in coding, the teacher circulated to check in and provide support, especially for student pairs who appeared to be struggling. Students spent an average total of 45 minutes coding over two class periods, before a "class model" was built.

Data Sources

Snaphots of student programs

The primary dataset for this study was obtained from the backend log files gathered automatically by the EcoMOD curriculum software and stored remotely on a University server in a PostgreSQL database. These log files contain, among other events, logged snapshots of the students' program (as in Figure 4, above) that are automatically generated each time the program is modified and run. To facilitate human coding, a script created PDF documents of these saved programs, showing each program in order, with timestamps, for each of the student pairs in the study. These program snapshots were then analyzed for this study.

Prior Programming Experience Survey

A survey of prior experience with programming was given to the students to see what associations prior experience might have with task performance. Students were asked which common apps, languages, or websites they had used, including code.org, Scratch, Minecraft (redstone, modding, or other coding), Arduino, Javascript, Gamemaker, Lego Mindstorm, Tynker, App Creation/App Inventor, and Python, or other. Students were also asked where they had previously learned programming or coding. Then students were asked "Overall, how much programming or coding would you say that you've done?" Students who responded with "none", "less than a week," or "a few weeks" were classified as beginners, while those who selected "less than a year" or "more than a year" were classified as experienced.

Teachers formed the pairs without taking into consideration students' prior programming experience. As a result, the pair combination of beginner, intermediate, and experienced students in programming varied in each class. In this sample, 14 pairs

were both beginners, 19 pairs were beginner/experienced, and 13 pairs were made up of two experienced students. Quality of programs was analyzed by prior experience level.

Rubric Design and Validation

Program analysis began by identifying each student pair's "final" model developed during independent programming. This was usually the last snapshot before the teacher built a model with the class. Occasionally, however, because pairs hadn't yet built a working model, they may have just cleared the workspace and started over when time was up. In those cases, beginning with the final snapshot before the class model, we worked backwards to identify the latest model that was closest to a working solution. This final model was then coded according to two different rubrics, as follows.

Functionality Rubric

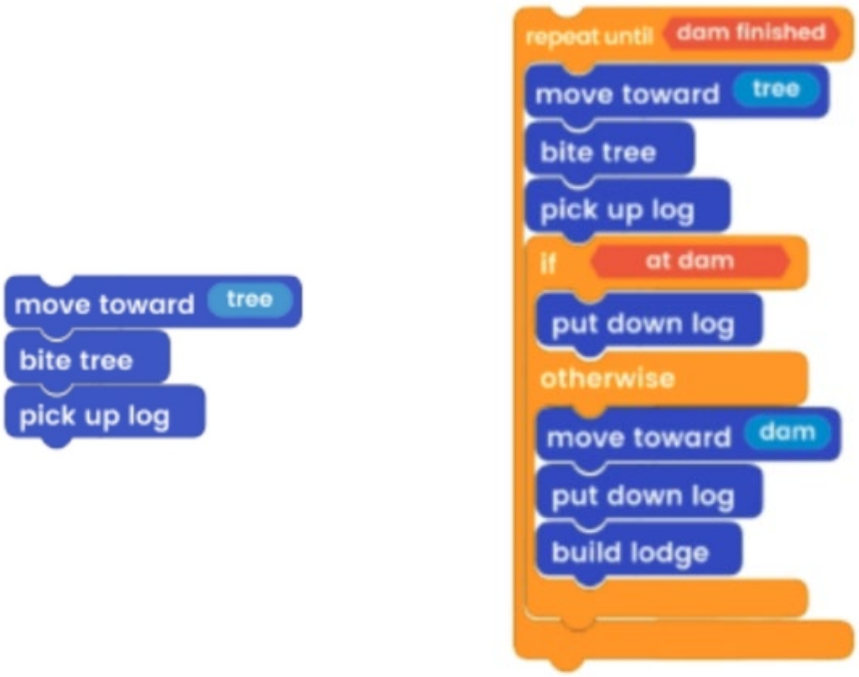
We first developed a *functionality rubric* to score how well the final model achieved the programming task: to build a dam and lodge while avoiding a predator. The rubric was based on the steps required of the beaver to complete the activity, as follows:

(1) Moving to a tree
(2) Biting tree until it became a log
(3) Picking up a log
(4) Bringing a log to dam and putting it down
(5) Finishing the dam (repeating the above sequence until the dam is finished)
(6) Building a lodge (executing the "build lodge" block once the dam is finished)
(7) Avoiding the predator (using the "near predator" conditional appropriately)

The task is strongly ordered, so that a program that achieves a step of the rubric must

have also been able to do all the previous tasks. The final model was therefore given a score of 0-7, based on the highest-numbered step the program was able to accomplish. The main goal of the task given to the students is building a lodge, for a score of 6, but pairs who were able to build a lodge were challenged to add predator-avoidance behavior to their models as well. Models that do so successfully are given a score of 7.

The first two working models in Figure 4 (above) score 6, and the third scores 7 because it also checks for a predator nearby. But when this rubric was applied to partially built models, we found that it was less accurate at evaluating how close students were to a correct solution. For example, a very simple sequence of three blocks might score 3, as in Figure 5a (below), because the workspace loops automatically, so eventually the beaver will reach a tree, bite it enough times, and pick up the resulting log. Other models in which the pair shows a developing understanding of loops and conditionals, as in Figure 5b, score 0 since the program doesn't work at all. It fails because the beaver will take one step toward the tree, but then as the program executes the "otherwise" actions, the beaver will take one step toward the dam. This forward-and-back "dance" was a challenging hurdle for many pairs.



a b

Figure 5. Examples of partially built models.

Conceptual Fluency Rubric

We then developed a *conceptual fluency* rubric that was more closely tied to student development of fluency in CT concepts. According to this rubric, the score is positively correlated with increasing complexity in the use of sequences, loops, and conditionals. The scores in this rubric represent milestones, similar to those in Franklin et al. (2017), as follows:

(1) Sequence: putting steps in order

(2) Loops (beginning) recognizing the need to repeat instructions, evidenced one of three ways:

   (a) Using the same block multiple times to repeat an action.

   (b) Using one block at a time in the workspace, to take advantage of built-in looping.

(c) Putting the whole sequence inside a “repeat until dam finished” block.

(3) Loops & Conditionals (basic: completing one subtask (e.g., move toward tree) using conditional blocks.

(4) Loops & Conditionals (developing): attempt to use multiple or nested conditional blocks.

(5) Loops & Conditionals (proficient): program almost successful, one error.

(6) Loops & Conditionals (advanced): Successful: fully working program.

(7) Advanced, with additional use of “predator nearby” conditional.

A score of 1, representing an understanding of sequence, involves simply putting the action blocks in the correct order, without any conditional blocks. A score of 2, beginning use of loops, includes three early strategies: (a) “step-by-step” repetition busing the same block multiple times in sequence (as in Figure 6a); (b) repetition “by hand,” through an interface quirk in which code can be edited without resetting the environment: Place a single “move toward tree” block in the workspace, click play and wait until the beaver reaches a tree, then remove it and place the new block “bite tree.” Then click play again, and since the beaver is already at the tree and it bites the tree, and so on, until the dam is complete; (c) placing the sequence of blocks inside a “repeat until dam finished” block (Figure 6b).

Scores of 3 through 6 represent stages of proficiency in the use of loops and conditionals, labeled as basic, developing, proficient, and advanced. Figure 6c is an example of model scoring 3, basic proficiency. Figure 5b, above, would score 4, developing proficiency.

a b c

Figure 6. More examples of partially built models.

Rubric Validation

Two raters, both members of the research team, applied the functionality rubric to all 47 pairs. The raters had prior familiarity with the software and programming task, and helped develop and refine the rubrics. Initial inter-rater reliability calculations resulted in a Cohen's Kappa of 0.631 (69% agreement). After meeting to discuss results, it was discovered that each coder had different interpretations of how to select the final program pairs used prior to being given the class model. After resolving the program selection procedure, the raters re-coded the programs with the functionality rubric with 100% agreement. Subsequently, the conceptual fluency rubric was applied in a similar fashion. Inter-rater reliability calculations for this rubric resulted in a Cohen's Kappa of 0.86 (90% agreement).

Case Study Comparison

In order to understand the differences found between scores using functionality and conceptual fluency rubrics, we selected two pairs to examine in-depth as a case study. One of these pairs was selected randomly from those that scored a 0 on their final

functionality rubric evaluation, but had previously run programs that made more progress. The second pair was randomly chosen from those that scored a 6 on the functionality rubric. In order to explore how their programs evolve over time, both rubrics were applied to *all program snapshots* of the two pairs, in contrast with the original analyses that considered only a final model from each pair. The two pairs were from two different schools involved in this study and thus had different teachers. Pair #1 was a beginner/experienced pair; pair #2 was made up of two beginners. While more examples would be beneficial for an in-depth analysis, the development of the current rubrics for use on final student programs is the focus of the work and deploying them reliably across all programs for all students was prohibitively time-consuming.

Results

Functionality Rubric

Results from applying the functionality rubric are shown in Figure 7. Scores were clustered at several points as follows. The most common score was 0, or "no tree", meaning that the program does not achieve even the basic objective of the beaver moving to a tree. The next most common stopping points were at 3, picking up a log, and at 6, completing the task by building a lodge successfully. Note that pairs scored at "avoid predator" were also able to successfully build the lodge, so 15 of the 47 pairs, or 32%, were fully successful, and 64% received a score of 3 or better.

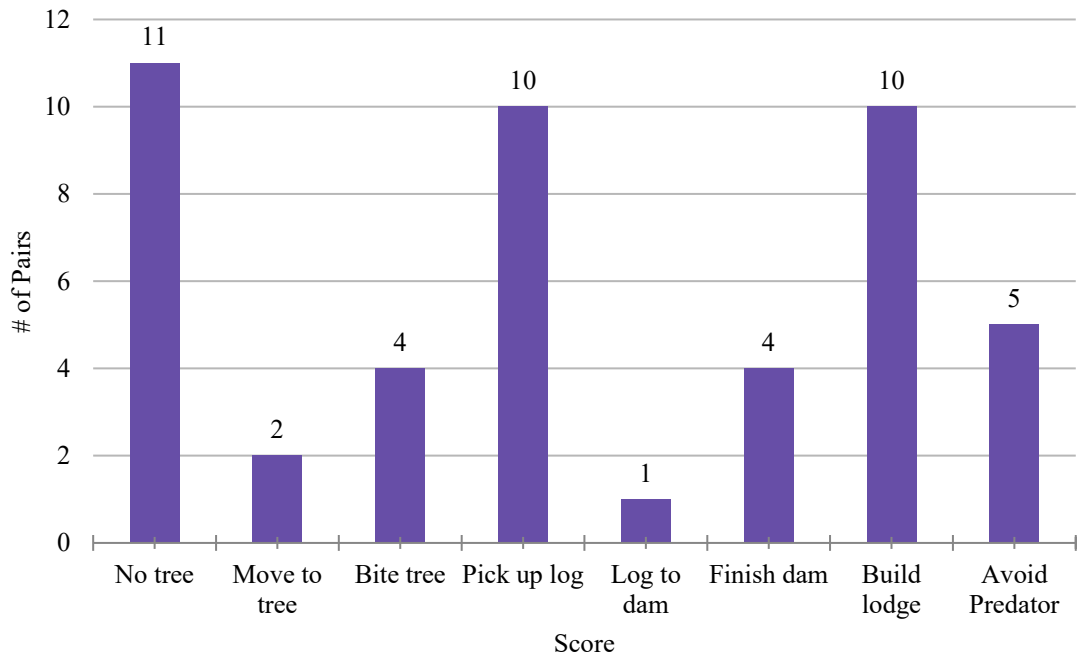


Figure 7. Count of all final functionality scores.

The 11 models that scored a 0 were those in which one or more errors kept even their "best" program from working at all. Due to the continuously looping nature of the programming tool, a single "move to tree" block is enough to get the beaver to move to a tree and thus a score of 1. Once at a tree, the addition of a "bite tree" and a "pick up log" block will rapidly get pairs to the next observed plateau at a score of 3. To bring the log back to the dam, however, represents a major increase in difficulty of the task, because of the forward-and-back dance of moving toward tree and toward dam. To identify which direction to move, conditional logic using multiple conditional statements must be used to successfully move on to the next scoring tier. Once that has been achieved, however, there are no new major conceptual barriers required to complete the task. Pairs that finished at "finish dam" instead of "build lodge" simply omitted a single block that actually built the lodge after the dam was complete.

No combination of prior programming experience (beginner/beginner, beginner/experienced, experienced/experienced) performed significantly better or worse on the functionality rubric based on an ANOVA model and confirmed with a Tukey's

honest significant difference test. Likewise, no classes performed significantly better or worse according to this scoring, and no use of specific conditionals (if, if/otherwise, etc.) was significantly associated with higher or lower functionality rubric scores. These findings will be explicated in the discussion.

Conceptual Fluency Rubric

Results of coding using the conceptual fluency rubric are shown in Figure 8. 19 pairs (40%) were scored as 4, developing understanding of loops and conditionals, having recognized the need for and attempted implementation of multiple conditionals. An additional four pairs scored as proficient, with one error away from success (e.g., omitting the "build lodge" block). Scores of 6 and 7 are the same for both rubrics: ten pairs showed advanced use of loops and conditionals, with fully successful models, and five pairs additionally showed an emerging understanding of events by checking for predators while also completing other tasks.

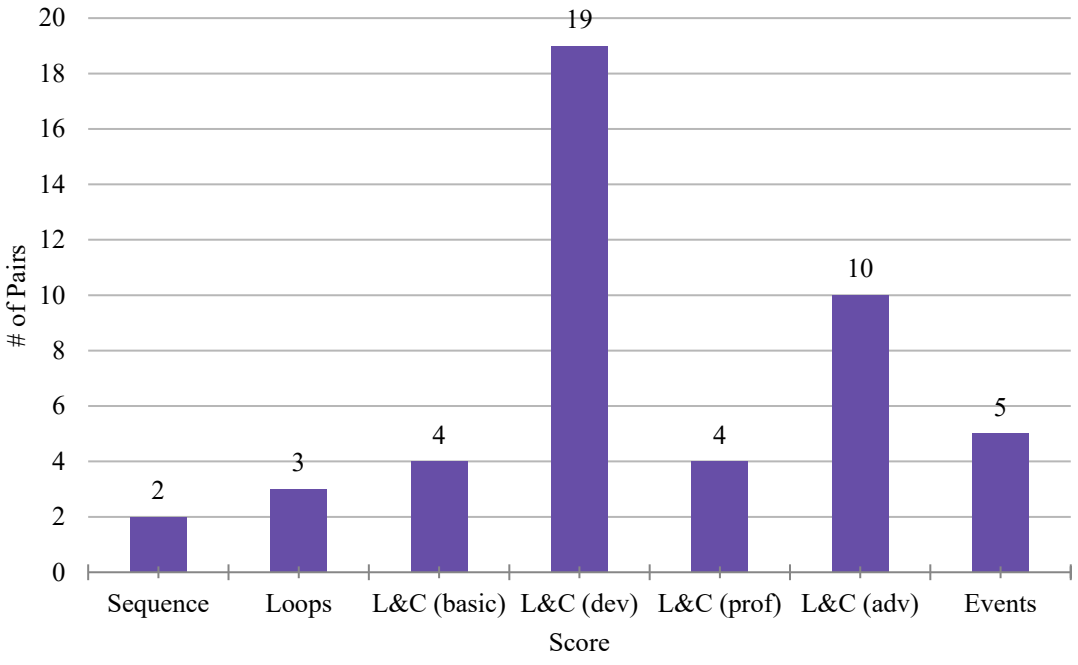


Figure 8. Count of all final conceptual fluency scores.

Similar to the functionality rubric results, no combination of prior programming experience performed significantly better or worse based on an ANOVA model and confirmed with a Tukey's honest significant difference test.

Case Study

For each of the two pairs selected for the in-depth case study (as described in the Methods section), we examined all program snapshots during the process of their programming activities over the two class periods. Each snapshot was coded with both rubrics. Plots of how their scores change over time are provided, as well as a narrative description of how their programs changed over time. In addition to applying the rubrics, the analysis also attends to process over time by identifying the transitions between program snapshots, e.g., adding to existing programs, revising/debugging existing code, or erasing their program and starting from scratch.

Figure 9 shows a graph of the coding of the snapshots for Pair #1. This pair had 26 program snapshots, and plotlines show the change in rubric score using both the functionality (in blue) and conceptual fluency (in red) rubrics.

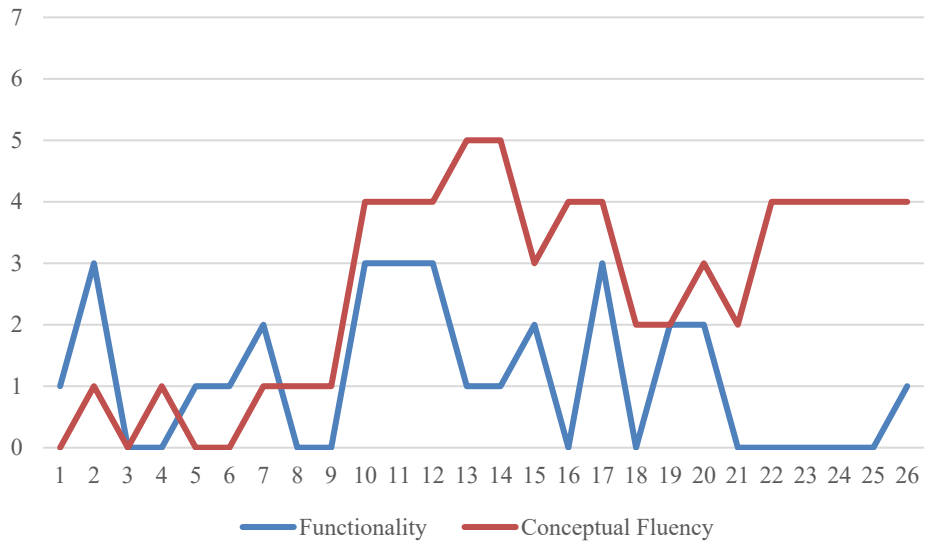


Figure 9. All functionality and conceptual fluency scores for Pair #1.

Pair #1 began by first running a program consisting solely of "move towards tree" then added the blocks to "bite tree" and "pick up log" in the next iteration. This second program easily accomplished the progress required for a 3 on the functionality rubric without any additional looping or conditionals. Once a log could be picked up, this pair then tried adding "move toward dam." This additional movement block in program 3 then hindered the beaver's ability to initially get to the tree, thus reducing the functionality score to 0 again. Pair #1 then tried several iterations of starting a new program and adding new blocks to in different orders during programs 3-9 to address this loss of functionality.

By program 10, this pair realized that "repeat until" blocks could solve this movement-related issue for both aspects of the task, thus raising their conceptual fluency score dramatically and regaining their prior functionality score. Programs 11-14 represent revisions to this program to fix mistakes (i.e., moving to "water" instead of "dam") and to try to implement the "put down log" behavior to the model. At program 13, the pair attempted to restructure their code to include more blocks inside "repeat until" conditions, but some errors in implementation reduced their functionality score again. Program 15 represented an attempt to start from scratch on a new day, this time utilizing "if-otherwise" blocks to achieve their desired behavior. Programs 16 through 20 represent their attempts at revising this alternate method with varying levels of success. Program 21 abandons this work in favor of a new program that attempts to include all blocks in a "repeat until dam finished" conditional. In subsequent programs, multiple "repeat until" and "if" statements are nested inside this conditional during a series of revision and adding moves, but the pair never achieved the level of functionality score they achieved with a simpler program.

Pair #2, in contrast, largely achieved all functionality and conceptual fluency goals for the curriculum (Figure 11). Programs 1 and 2 established the basic sequence needed to move and bite but were missing the "tree" portion of "move toward tree." In program 3, a "repeat until at tree" conditional was added, and program 4 added behavior so that the beaver could successfully chop down a tree and pick up a log. In program 6, a second "repeat until" conditional was used to move the beaver back to the dam with a log. Program 7 showed the addition of a "put down log" command, and revisions made by program 9 allowed the beaver to successfully build the dam. This program was missing a "build lodge" command, thus limiting it from achieving a rating of 6 on the functionality rubric. After some small-scale revisions, Program 12 includes the entire program in a "repeat until dam finished" block. With the addition of a "build lodge" command after this loop in program 14, the program is thus able to successfully complete the task. An erroneously placed "move to tree" block was added in program 15 that hampered performance but was removed in the next iteration.

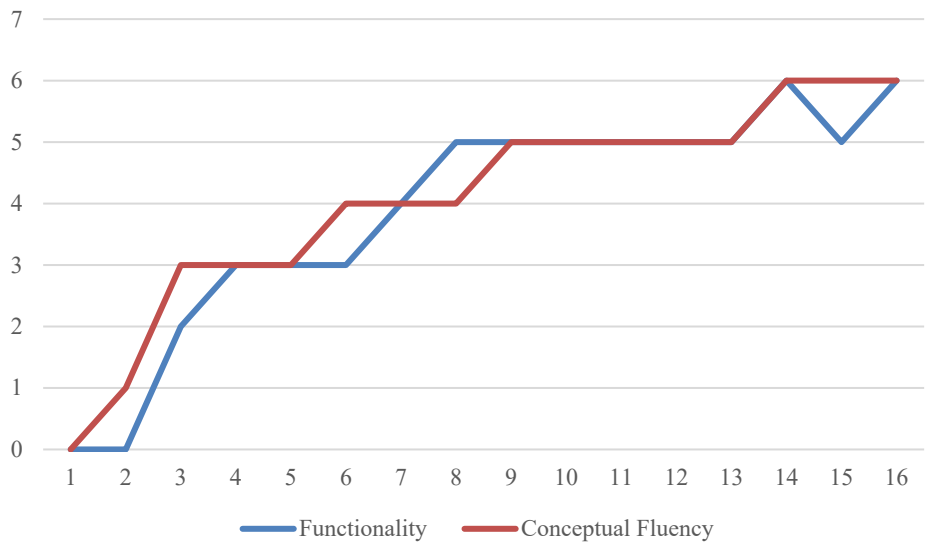


Figure 11. All functionality and conceptual fluency scores for Pair #2.

Discussion

Overview

With no prior instruction, fifteen of the 47 pairs were able to construct a fully functional computational model of the beaver over approximately 45 minutes spread over two class periods, and an additional four pairs were able to complete the task except the final step of building a lodge. Using the functionality rubric, we found that pairs had an average score of 3.38, with significant hurdles moving to multiple conditionals and loops that often resulted in a score of 0 for partially complete models.

Assessment using the conceptual fluency rubric found that pairs that were not able to finish in the limited amount of time allotted still demonstrated learning of CT concepts such as sequence, loops and conditionals. All pairs showed an understanding of sequence, and almost all pairs were able to at least engage in use of multiple or nested conditionals, even when some encountered hurdles in getting them to work together correctly.

Results from both rubrics revealed no significant differences between beginner/beginner, beginner/experienced, and experienced/experienced pairs. This was a surprising finding, possibly explained by having a rather small sample, but we can also posit other possible explanations. The information on experience level was self-reported by these young students in the pre-survey, so it is possible that some misestimated the amount of time they spent programming in the past. It also seems possible that time spent in coding activities may not be the most relevant criterion, particularly considering the different types of previous experience (e.g., Minecraft redstone versus Scratch). These coding activities may have different impacts on students' abilities to generate scientific models in this curriculum, as the task might have been different enough from their prior experiences that the transfer was quite low.

The case study showed that both pairs' scores largely increased over time according to the conceptual fluency rubric as pairs grew in their use of loops and conditionals. The functionality score for pair #1, however, repeatedly rose and fell as pair #1 attempted different strategies with conditionals and wasn't able to get their code to work.

While not a focus of this research, looking at the longitudinal record of programs over time in this way also offers the opportunity to look at use of CT practices such as experimenting and iterating, testing and debugging (Brennan & Resnick, 2012). As noted in the literature, these strategies distinguish experts from novices, but are also the kinds of strategies that lead to novices becoming more effective in learning programming (Robins et al., 2003). Case study pair #1 exhibited a pattern of throwing out code that wasn't working and starting from scratch, while pair #2 iteratively added to, tested, and revised their model. Pair #2, then, could be said to be more effective in their use of CT practices, though, as noted above, pair #1 was a beginner-experienced pair and pair #2 were both beginners according to the programming pre-survey.

Assessment via functionality rubric

The first research question for this study asks what aspects of computational thinking can be assessed with a functionality rubric. This type of rubric had several advantages, namely that it provided a quick overview of student progress to see what curricular milestone pairs reached. This style of rubric is easier to understand and apply as it does not require in-depth knowledge of CT principles or learning progressions. Since the elementary teachers using this curriculum are not necessarily familiar with CT or computer science education, the scoring levels of this rubric are more easily understood by our target users.

The functionality rubric is more linked with the curricular goal of scientific modeling, and the rubric is constrained in a similar fashion to the programming tool itself. There was only one possible series of steps by which the beaver could build the dam, though there were a range of programming variations to implement those steps. A functionality rubric would be more useful to assess the scientific accuracy of the model if the code blocks were more flexible. The beaver is seen in the 3D virtual world acting a certain way and students are shown videos of real beaver behavior. These observations form the ground truth of what a beaver functionally can and cannot do, and students are instructed to mimic these behaviors in their model. If pairs could change the beaver to carry multiple logs, leap to a tree in one step, or attack the wolf, a rubric like this could assess how closely the computational model aligns with the ground truth seen in the 3D virtual world.

The results of this rubric could also inform the design of curriculum and suggest new ways of supporting learners engaged in computational modeling. By seeing when certain roadblocks happen (i.e., getting stuck trying to get the log to the dam), more supports could be added for teachers to deal with these common issues. These findings could also suggest when the class model should be introduced so that the teacher can provide support when most pairs are at a similar stage. Conversely, this rubric had the notable disadvantage of over-assigning pairs a score of 0. Even small errors in a mature program may have meant a final score of 0, thus failing to consider progress made to that point. Other very simple models resulted in higher scores since a score of 3 on the rubric could be achieved with a few simple blocks and no conditionals.

Assessment via a conceptual fluency rubric

The second research question likewise asked what aspects of computational thinking can be assessed with a conceptual fluency rubric. Unlike the functionality rubric, this

rubric provides more nuanced information about student progress towards understanding CT. Students whose final model scored a 0 in the functionality rubric often scored higher here as they had made progress towards using conditionals and understood sequence at a basic level. We are able to see where they got stuck by looking at this rubric, and could identify the common hurdle of the jump to needing multiple conditional statements. Even if errors prevent programs from being fully successful, this rubric is a much better assessment of CT with a focus on learning sequences, loops, and conditionals.

This method of scoring programs can also drive curriculum development and teacher-facing support. Modeling how to properly use multiple conditionals to achieve a goal in the programming tool may help teachers see the types of errors students are making and help them understand how to intervene. Eventually, automated supports could also be included in the software depending on what error messages are being triggered.

Ramifications for the curriculum

The third research question asked how this assessment can inform curriculum design. The stages of program development exhibited by student pairs, as identified in the conceptual fluency rubric, were used to inform revisions to the design of the curriculum and professional development materials.

The computational modeling curriculum lessons were revised to address a number of issues identified through this analysis. First, a discussion of the programming blocks in the lesson plan was added to address student misconceptions. For example, some students didn't use "build lodge" because they didn't understand it, so "lodge" was introduced as a vocabulary word, as the home that beavers build in the middle of the pond once it is formed by the dam. Also added was a discussion of differences

between similar blocks, e.g, What does it mean for the beaver to “have log” vs being “at log?” What is the difference between “move toward water” and “move toward dam,” and when would you use each one?

Lessons during student programming were also modified to support CT processes. Recommendations were added for the teacher to model debugging strategies with the class, including using the step function to see how each block in the code is enacted, and looking at error messages to see how they can be used to debug (e.g., “I’m not at a tree so I can’t bite a tree” suggests that the “bite tree” block should only be executed when the beaver is at the tree). The lesson time was extended so that students would have a chance to share their models with each other or the class – especially including opportunities to share a place where they got stuck, in order to get feedback from other students as well as the teacher.

Recommendations and tips for programming were also integrated into the teacher-facing materials, to help teachers understand and explain to students about sequences, looping, and conditionals, and to recognize common errors made by students. For each common error, the materials provide ways to identify the error, why it is an error, and ways to guide students in solving it. For example, one common error addressed in the teacher professional development materials relates to students repeating blocks instead of using a conditional (Figure 12). The “how you can help” section includes questions teachers can ask students to provide hints and encouragement for the student to figure out the next step. The step-by-step procedure will allow teachers without much programming experience to be able to support students with more confidence.

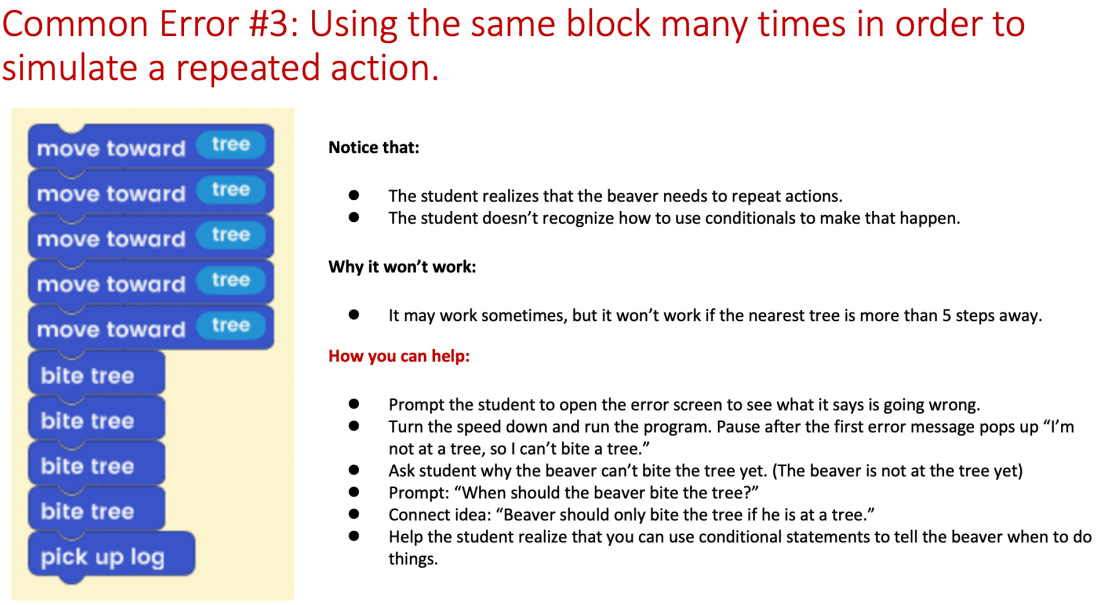


Figure 12: Example page for Teachers' Guide on helping learners during programming activity.

Limitations and Future Work

This study assesses student use of computational concepts like sequence, loops and conditionals, but it is not a direct measure of how well they understood these concepts. Further exploration of student learning in EcoMOD via triangulation with traditional CT assessment tests or structured interviews would help explore the extent of student CT understanding. This activity also only involves a single programming task with a specific goal. Now that the educational value of EcoMOD has been established (Dickes, et al., 2019; Jeon et al., 2020), a broader curriculum with increased opportunities for modeling other types of creatures and systems may give students a chance to showcase more CT learning. Furthermore, we note that human coding of program snapshots is laborious, especially when examining partially built models that may contain errors. While necessary for eventual automation, these scoring procedures introduce a significant barrier to this type of assessment.

In addition to lowering the sample size, the use of pairs sharing one device lead to issues of not knowing how equal contributions were within pairs. The resultant pair-level scores left us unable to evaluate CT fluency by individual students within pairs. Additionally, conversations within and between pairs were common as students sat in clusters. These informal discussions are not captured and may have led to groups learning from each other. Similarly, we are unable to tell when or if teachers intervened or suggested a modification, versus pairs deciding on a new direction independently. Future work can examine classroom video to explore these interactions, to see what informal conversations looked like during the programming activity, as well as examine teacher variation in implementation.

One important area for future research would be to consider how teachers might be able to use these rubrics themselves, or to develop similar rubrics for different contexts. For this study, the research team both developed and applied the rubrics to student programs. The functionality rubric is relatively straightforward to apply; one can determine how well the program achieves the task by running it. The conceptual fluency rubric, however, requires more familiarity with coding in order to identify milestones in fluency with CT concepts, so its use by teachers would likely involve significant preparation and support.

Another interesting extension of this work could also include adapting these rubrics into automated scoring tools similar to programs like Dr. Scratch (Moreno-León et al., 2015). Further, coding of pairs' entire sequence of programs, as done for the two case study pairs in this research, might be automated and analyzed for data about student CT practices. Automated analyses might be used to identify sequences that demonstrate more mastery in practices such as iteration, testing, and debugging, and perhaps inform the design of scaffolding to guide students in these practices.

Conclusion

This study addresses CT assessment of programming artifacts constructed by student pairs to achieve a specific agent-based modeling task. We first developed a rubric based on functionality, and then, having identified limitations of this perspective, a second rubric based on identifying evidence of conceptual fluency. We found that each methodology provided a different lens on the data. The functionality rubric assessed how well the program achieved the task, but was less successful at correctly identifying CT progress in programs that were partially correct. A rubric for conceptual fluency, in contrast, was more able to recognize stages of development in use of CT concepts.

The assessment of student CT understanding with EcoMOD found that, with no direct instruction on programming, 3rd grade student pairs with minimal prior programming experience were able to make progress in using computational concepts of sequencing, loops, and conditionals. In related papers, and in our ongoing research, we examine how the design of the intervention, using a visual, block-based interface and a small set of custom, domain-specific building blocks, made the task more accessible, as did the embodiment activities in the 3D world in which students could take on the role of the beaver building the dam. (Dickes et al., 2019).

This research represents an exploration of CT assessment applied to a specific context: agent-based computational models for 3rd grade ecosystem science learning. The assessment rubrics described in the paper are based on two methodologies, functionality and computational fluency, which we suggest can be applied broadly to CT-based tasks beyond this context, to other age levels, and in different contexts. Functionality assessment focuses attention on progress towards a programming goal, while computational fluency assessment supports evaluation of milestones in use of CT concepts. It is our hope that the findings related to these two assessment strategies may

inform future research and development of assessment of student programming artifacts beyond this context, and may encourage future research that considers the affordances of different dimensions of rubric-based CT assessment. We have added to the future directions section a suggestion that future work might look into how teachers might be able to use rubrics.

Acknowledgements

The authors would like to express our appreciation to Amanda Dickes for her intellectual contributions to this work. This material is based upon work supported by the National Science Foundation under grant No. DRL-1639545. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Adams, C., Cutumisu, M., & Lu, C. (2019, March). Measuring K-12 computational thinking concepts, practices and perspectives: An examination of current CT assessments. In Society for Information Technology & Teacher Education International Conference (pp. 275-285). Association for the Advancement of Computing in Education (AACE).

Aksit, O., Wiebe, E.N. (2020). Exploring Force and Motion Concepts in Middle Grades Using Computational Modeling: A Classroom Intervention Study. J Sci Educ Technol 29, 65–82

Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., & Koller, D. (2014). Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. Journal of the Learning Sciences, 23(4), 561-599.

Brennan, K., & Resnick, M. (2012, April). New frameworks for studying and assessing the development of computational thinking. In Proceedings of the 2012 annual meeting of the American educational research association, Vancouver, Canada (Vol. 1, p. 25).

Buitrago Flórez, F., Casallas, R., Hernández, M., Reyes, A., Restrepo, S., & Danies, G. (2017). Changing a generation's way of thinking: Teaching computational

thinking through programming. Review of Educational Research, 87(4), 834-860.

Chen, G., Shen, J., Barth-Cohen, L., Jiang, S., Huang, X., & Eltoukhy, M. (2017). Assessing elementary students' computational thinking in everyday reasoning and robotics programming. Computers & Education, 109, 162-175.

Dagiene, V., & Stupuriene, G. (2016). Bebras--A Sustainable Community Building Model for the Concept Based Learning of Informatics and Computational Thinking. Informatics in education, 15(1), 25-44.

Dickes, A.C., Kamarainen, A., Metcalf, S.J., Gun-Yildiz, S., Brennan, K., Grotzer, T., & Dede, C. (2019) Scaffolding Ecosystems Science Practice by Blending Immersive Environments and Computational Modeling, British Journal of Educational Technology.

Dickes, A. C., Sengupta, P., Farris, A. V., & Basu, S. (2016). Development of mechanistic reasoning and multilevel explanations of ecology in third grade using agent-based models. Science Education, 100(4), 734-776.

Dwyer, H., Boe, B., Hill, C., Franklin, D., & Harlow, D. (2013). Computational thinking for physics : Programming models of physics phenomenon in elementary school. Physics Education Research Conference, 133–136. doi:10.1119/perc.2013.pr.021

Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., Weintrop, D. & Harlow, D. (2017, March). Using upper-elementary student performance to understand conceptual sequencing in a blocks-based curriculum. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (pp. 231-236). ACM.

Giere, R.N. (1988). Explaining science: A cognitive approach. Chicago, IL: University of Chicago Press.

Grover, S., & Basu, S. (2017, March). Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education (pp. 267-272).

Grover, S., & Pea, R. (2018). Computational Thinking: A competency whose time has come. Computer science education: Perspectives on teaching and learning in school, 19.

Hutchins, N.M., Biswas, G., Maróti, M., Lédeczi, Á., Grover, S., Wolf, R., Blair, K.P., Chin, D., Conlin, L., Basu, S. & McElhaney, K. (2020). C2STEM: a System for Synergistic Learning of Physics and Computational Thinking. Journal of Science Education and Technology, 29(1), pp.83-100.

Israel, M., Pearson, J. N., Tapia, T., Wherfel, Q. M., & Reese, G. (2015). Supporting all learners in school-wide computational thinking: A cross-case qualitative analysis. Computers & Education, 82, 263-279.

Jeon, S., Metcalf, S., Dickes, A. & Dede, C. (2020). Elementary teacher perspectives on a blended computational modeling and ecosystem science curriculum. In D. Schmidt-Crawford (Ed.), Proceedings of Society for Information Technology & Teacher Education International Conference (pp. 46-55).

Klopfer, E., Yoon, S., & Um, T. (2005). Teaching complex dynamic systems to young students with StarLogo. Journal of Computers in Mathematics and Science Teaching, 24(2), 157-178.

Lane, H. C., & VanLehn, K. (2005). Intention-based scoring: an approach to measuring success at solving the composition problem. In ACM SIGCSE Bulletin (Vol. 37, No. 1, pp. 373-377). ACM.

Lee, I., Martin, F., & Apone, K. (2014). Integrating computational thinking across the K--8 curriculum. ACM Inroads, 5(4), 64-71.

Lehrer, R. (2009). Designing to develop disciplinary dispositions: Modeling natural systems. American Psychologist, 64(8), 759.

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.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. ACM Transactions on Computing Education (TOCE), 10(4), 1-15.

Moreno-León, J., Robles, G., & Román-González, M. (2015). Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking. RED. Revista de Educación a Distancia, (46), 1-23.

Nersessian, N. (2008). Model-based reasoning in scientific practice. In Teaching scientific inquiry (pp. 57-79). Brill Sense.

NGSS Lead States (2013). Next Generation Science Standards: For States, By States. Washington DC. The National Academies Press.

Papert, S., & Harel, I. (1991). Situating constructionism. Constructionism, 36(2), 1-11.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. and Kafai, Y. (2009). Scratch: programming for all. Communications of the ACM, 52(11), pp.60-67.

Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., & Franklin, D. (2017, August). K-8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In Proceedings of the 2017 ACM Conference on International Computing Education Research (pp. 182-190). ACM.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education, 13*(2), 137-172. doi:10.1076/csed.13.2.137.14200

Román-González, M., Moreno-León, J., & Robles, G. (2017, July). Complementary tools for computational thinking assessment. In Proceedings of International Conference on Computational Thinking Education (CTE 2017), S. C Kong, J Sheldon, and K. Y Li (Eds.). The Education University of Hong Kong (pp. 154-159).

Román-González, M., Pérez-González, J. C., & Jiménez-Fernández, C. (2017). Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. Computers in Human Behavior, 72, 678-691.

Ryu, S., Han, Y., & Paik, S. H. (2015). Understanding co-development of conceptual and epistemic understanding through modeling practices with mobile internet. Journal of Science Education and Technology, 24(2-3), 330-355.

Sengupta, P., Dickes, A., Farris, A. V., Karan, A., Martin, D., & Wright, M. (2015). Programming in K-12 science classrooms. Communications of the ACM, 58(11), 33-35.

Sengupta, P., Kinnebrew, J. S., Basu, S., Biswas, G., & Clark, D. (2013). Integrating computational thinking with K-12 science education using agent-based computation: A theoretical framework. Education and Information Technologies, 18(2), 351-380.

Shute, V. J., Sun, C., & Asbell-Clarke, J. (2017). Demystifying computational thinking. Educational Research Review, 22, 142-158.

Tang, X., Yin, Y., Lin, Q., Hadad, R., & Zhai, X. (2020). Assessing computational thinking: A systematic review of empirical studies. Computers & Education, 103798.

Troiano, G. M., Snodgrass, S., Argimak, E., Robles, G., Smith, G., Cassidy, M., Tucker-Raymond, E., Puttick, G., & Harteveld, C. (2019). Is my game OK Dr. Scratch?: Exploring programming and computational thinking development via metrics in student-designed serious games for STEM. In Proceedings of the 18th ACM International Conference on Interaction Design and Children (pp. 208-219). ACM.

Villamor, M. M. (2020). A review on process-oriented approaches for analyzing novice solutions to programming problems. Research and Practice in Technology Enhanced Learning, 15(1), 1-23.

Wagh, A., Cook-Whitt, K., & Wilensky, U. (2017). Bridging inquiry-based science and constructionism: Exploring the alignment between students tinkering with code of computational models and goals of inquiry. Journal of Research in Science Teaching, 54(5), 615-641.

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. Journal of Science Education and Technology, 25(1), 127-147.

Weintrop, D., Hansen, A. K., Harlow, D. B., & Franklin, D. (2018, August). Starting from Scratch: Outcomes of early computer science learning experiences and implications for what comes next. In Proceedings of the 2018 ACM Conference on International Computing Education Research (pp. 142-150).

Weintrop, D., & Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. ACM Transactions on Computing Education (TOCE), 18(1), 1-25.

Werner, L., Denner, J., Campe, S., & Kawamoto, D. C. (2012, February). The fairy performance assessment: measuring computational thinking in middle school. In Proceedings of the 43rd ACM technical symposium on Computer Science Education (pp. 215-220).

Wing, J.M. (2006). Computational thinking. Communications of the ACM, 49(3), 33-35.

Worsley, M., & Blikstein, P. (2013, July). Programming pathways: A technique for analyzing novice programmers' learning trajectories. In International Conference on Artificial Intelligence in Education (pp. 844-847). Springer, Berlin, Heidelberg.

Yoon, S. A., Anderson, E., Koehler-Yom, J., Klopfer, E., Sheldon, J., Wendel, D., Schoenfeld, I., Scheintaub, H., Oztok, M., & Evans, C. (2015). Design Features for Computer-Supported Complex Systems Learning and Teaching in High School Science Classrooms In Lindwall, O., Häkkinen, P., Koschman, T. Tchounikine, P. Ludvigsen, S. (Eds.) (2015). Exploring the Material Conditions of Learning: The Computer Supported Collaborative Learning (CSCL) Conference 2015, Volume 1. Gothenburg, Sweden: The International Society of the Learning Sciences.