Informatics in Education, 2020, Vol. 19, No. 4, 611–640

DOI: 10.15388/infedu.2020.27

Assessing 4th Grade Students' Computational Thinking through Scratch Programming Projects

Janne FAGERLUND[1]*, Päivi HÄKKINEN[2],
Mikko VESISENAHO[1], Jouni VIIRI[1]

[1]*Department of Teacher Education, University of Jyväskylä, Jyväskylä, Finland*
[2]*Finnish Institute for Educational Research, University of Jyväskylä, Jyväskylä, Finland*
e-mail: {janne.fagerlund, paivi.m.hakkinen, mikko.vesisenaho, jouni.p.t.viiri}@jyu.fi

Received: June 2020

Abstract. Computational thinking (CT) has been introduced in primary schools worldwide. However, rich classroom-based evidence and research on how to assess and support students' CT through programming are particularly scarce. This empirical study investigates 4th grade students' (N = 57) CT in a comparatively comprehensive and fine-grained manner by assessing their Scratch projects (N = 325) with a framework that was revised from previous studies to aim towards enhancing CT. The results demonstrate in detail the various coding patterns and code constructs the students programmed in assorted projects throughout a programming course and the extent to which they had conceptual encounters with CT. Notably, the projects indicated CT diversely, and the students altogether encountered dissimilar areas in CT. To target the acquisition of CT broadly, manifold programming activities are necessary to introduce in the classroom. Furthermore, we discuss the possibilities of applying the assessment framework employed herein to support CT education through Scratch in classrooms.

Keywords: computational thinking; Scratch; assessment; primary school; education.

Introduction

Computational thinking (CT) has been increasingly incorporated in primary schools across the world often by means of graphical programming (Bocconi *et al.*, 2018; Mannila *et al.*, 2014). Using Scratch to design interactive games, animations, and stories that are thematically connected to different curricular areas is especially popular among the age group (Garneli *et al.*, 2015; Moreno-León *et al.*, 2017). Being a new topic in primary education, however, CT involves areas that necessitate research-based pedagogical knowledge. In particular, it is vital to more intricately spotlight what CT students

* Corresponding author. Postal address: Ruusupuisto, P.O. Box 35, 40014 University of Jyväskylä, Finland. e-mail: janne.fagerlund@jyu.fi Tel. +358408054711

can gain in various ways and how their CT learning could be pedagogically supported while programming (Lye and Koh, 2014; Shute *et al.*, 2017). On a related note, although previous literature (e.g., Barr and Stephenson, 2011) describes what kinds of skills and knowledge CT involves, the term still has no universally accepted depiction. Adopting a relatively extensive view of CT (as opposed to an overly concise one merely embodying tool-specific programming skills) enables investigating skills that are potentially transferable across problem-solving domains. To that end, this study investigates primary school students' CT deeply based on Scratch projects they programmed in naturalistic classroom situations. In the process, we also aim to develop ways to support students' learning of CT in this context.

These aspirations led us to "assessment for learning", in particular, formative assessment, which can support students' learning performance and their beliefs about their own capabilities (Black and Wiliam, 1998; 2009). When outlining CT rather extensively, though, various types of programming contents in Scratch can indicate CT (Seiter and Foreman, 2013). Our review of relevant studies shows that although several assessments of CT in Scratch projects exist, they mostly cover contents that indicate partial areas in CT, such as its certain core concepts or principles in particular learning scenarios.

The objective of this study is to gain rich empirical insight of 4th grade students' CT by assessing Scratch projects that they designed during a programming course. In order to assess the students' CT extensively through their Scratch projects and set a stage to facilitate known learning benefits in formative assessment in the classroom in the future, we were encouraged to build on existing works to revise an especially profound assessment framework. This article reports on the preparatory use of the framework in assessing programming contents and indicative CT in the students' different kinds of Scratch projects in a comparatively comprehensive and fine-grained manner. By comprehensiveness, we refer to the wide-ranging categorization embodying what students can learn in CT and how manifold programming contents indicate CT. By fine granularity, we refer to the way of systematically analyzing small-scale programmatic evidence in Scratch projects for advantages in research and learning-support alike. We evaluate and discuss the significance of the attained evidence in CT education and the next steps of developing formative assessment of CT in schools.

Assessing CT in Scratch Projects

Positioning CT in Primary Education

The term computational thinking (CT) was popularized by Jeannette Wing (2006; 2011) as "the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer – human or machine – can effectively carry out." CT provides competencies for adapting to the digitalized world and solving problems across disciplines by applying computational tools, models, and ideas (Denning

and Tedre, 2019). Broadly viewed as a competence that, apart from mere programming skills, involves broader concepts and practices, such as algorithms, data, and problem decomposition, which describe manifold skills and areas of understanding that are expected to transfer to different problem-solving domains (Angeli *et al.*, 2016; Barr and Stephenson, 2011; Csizmadia *et al.*, 2015; Grover and Pea, 2018; Hsu *et al.*, 2018; Shute *et al.*, 2017). Amid continuing efforts to conclusively capsulize the exact nature of CT, we define the kinds of skills and knowledge it can involve in a relatively inclusive rather than overly condensed manner.

In several countries, CT or its proximal topics (e.g., informatics, computer science, programming) are integrated in the learning of different curricular areas (Heintz *et al.*, 2016). At the primary school level, block-based programming has been an especially popular way to promote CT (Grover and Pea, 2013). Meaningful learning can occur in multidisciplinary project-based settings in which students have autonomy regarding, for instance, how they learn (Lonka, 2018). Scratch has been often used in teaching and learning practice, for example, through creative game design, storytelling, or animation while the substance of other curricular areas is being processed (Garneli *et al.*, 2015; Moreno-León *et al.*, 2017).

The focus of this study is in assessment for learning, which aims to promote learning rather than merely rank or certify it. In particular, the processes associated with formative assessment can support classroom learning (Black and Wiliam, 2009). Programming is cognitively complex, and support for learning CT through programming is vital for making learning more effective (Lye and Koh, 2014). However, CT involves several aspects, and it should be assessed from several entry points (Grover *et al.*, 2017). Earlier research has indicated that assessing students' Scratch projects is among essential entry points because programmed projects are rich, concrete, and contextualized approximations of the students' conceptual encounters with CT (Brennan and Resnick, 2012; Román-González *et al.*, 2019; Seiter and Foreman, 2013). Therefore, whilst adopting a comparatively inclusive view of CT, in the following sections we review previous studies on the assessment of students' Scratch projects from the viewpoint three formative assessment processes: what to teach and learn (i.e., clarifying learning objectives), estimating students' current level of understanding, and providing relevant feedback (Black and Wiliam, 2009).

Assessment for Learning in Scratch

What to Teach and Learn

CT can be concretized in programming through core educational principles, such as "algorithm control structures", "parallel execution", and "Boolean logic". These principles can be manifested in Scratch in at least three categories: code constructs, coding patterns, and other programming contents. (Fagerlund *et al.*, 2020.)

Code constructs, akin to language primitives, such as those for controlling the flow in programs (i.e., "sequence", "conditional", "loop") can be observed directly as Scratch blocks. For instance, Moreno-León *et al.* (2015) used Dr. Scratch, an automated analysis tool, to examine the presence of sequences of blocks, "repeat" blocks and "if" blocks in

Scratch projects. Other studies (Basu, 2019; Franklin *et al.*, 2013; 2017; Maloney *et al.*, 2008; Meerbaum-Salant *et al.*, 2013) additionally examined projects manually for certain blocks representing constructs, such as "initialization" and "coordination."

Additionally, akin to classes, projects can comprise *coding patterns*: semantically meaningful combinations or templates of constructs that achieve specific functionalities (e.g., animation of size). For instance, the Progression for Early Computational Thinking (PECT) model assesses project-wide patterns, such as "Maintain score" and "User interaction", which incorporate different types of templates that can be programmed by combining specific code constructs in specific ways (Seiter and Foreman, 2013). Franklin *et al.* (2013; 2017), Ota *et al.* (2016), and Seiter (2015) examined similar patterns to those in PECT, such as "Count up score" and "Multi-sprite synchronization."

Concerning the more social elements of programming and metaprogrammatic elements, projects can also contain *other programming contents*, such as "project use instructions" and the "appropriate naming of sprites" (Basu, 2019; Funke *et al.*, 2017; Wilson *et al.*, 2012).

Manipulating such contents in Scratch simultaneously fosters and demonstrates CT (Brennan and Resnick, 2012). Learning goals and intentions (Black and Wiliam, 2009) regarding students' skills and areas of understanding in CT can thus be clarified indirectly as meaningful and reasonably demanding contents for students to creatively design in projects. Aggregating the various contents distributed among previous studies and assessment frameworks can establish systematic and comprehensive (albeit not necessarily all-encompassing) coverage of CT-fostering programming contents in Scratch.

Estimating Current Level of Understanding

Students' learning can be enhanced by guiding them to perceive a gap between set learning goals and their own present skills or understanding. Information regarding this gap can be generated by students, peers, or instructors. (Black and Wiliam, 1998.) In CT, evidence that points towards students' skills and understanding can be elicited by examining what they have programmed in their projects (Grover and Pea, 2013; Román-González *et al.*, 2019; Seiter and Foreman, 2013). More tangibly, programmed contents in projects indicate conceptual encounters with CT's core educational principles. However, it is crucial that learning tasks generate and display relevant evidence of learning (Black and Wiliam, 1998). Evincing CT through programmed projects can be risky for two reasons.

First, static artifacts are not direct measurements of thinking. In particular, block-based programming environments can present validity concerns, as students can drag and drop blocks without knowing what they are doing (Lye and Koh, 2014). Hence, observational methods should involve rigorously considering the circumstance of the evidence. Analyzing contents primarily through coding patterns improves validity by ensuring that the implemented blocks achieve semantically meaningful computational models (Seiter and Foreman, 2013).

Second, Scratch enables various project design opportunities: projects can be designed in different genres, such as animation, game, and story (Maloney *et al.*, 2010),

which typically contain different programmatic characteristics and, respectively, indications for CT (Moreno-León *et al.*, 2017). Programming can also be effectuated through activities such as remixing or debugging preexisting contents or designing something new (Lee *et al.*, 2011). Moreover, individual experiences may vary when students are activated as instructional resources for each other, for instance, in pair programming. Such contextual factors influencing what constitutes relevant evidence of learning may increase the more students are activated in owning their own learning. (Black and Wiliam, 2009.) In summary, to advocate the relevance of the examined contents, it is preferable to consider the context of the contents (e.g., recognizing the learning assignment) and interpret their semantical significance rather than merely technical one.

Providing Feedback

After a gap between learning goals and the students' current knowledge has been highlighted, the students are guided to take action to close that gap (Black and Wiliam, 1998). While students program, meaningful and authentic feedback can be provided with respect to the contents in their projects. The feedback should facilitate the correction of specific errors or poor strategies with suggestions on how to improve the work (e.g., by providing scaffolding), based on individual progress toward achieving goals (Black and Wiliam, 1998). In this respect, alternatively to evincing the most proficient segments in students' projects, assigning scores, and providing generic feedback (see Moreno-León *et al.*, 2015; Seiter and Foreman, 2013), project contents can be examined "micro-programmatically" (e.g., Vihavainen *et al.*, 2013).

In Scratch, the scripts of a project can comprise individually instantiated coding patterns and their underlying code constructs. For instance, sprites' properties, such as location or size, are animated distinctly from one another (see Franklin *et al.*, 2013, 2017; Meerbaum-Salant *et al.*, 2013). Such fine-grained evidence, that is, specific micro-programmatic project parts, in fact, benefits all three formative assessment processes: operating as meaningful learning goals for students to program in projects, semantically meaningful evidence of their understanding, and meaningful targets for feedback. However, examining instantiated coding patterns is an overlooked analytical approach when applied to CT-fostering programming contents comprehensively and systematically.

The Current Study

The purpose of this study was to gain rich empirical insight of 4th grade students' (N = 57) CT by assessing the programming contents in Scratch projects (N = 325) that they designed during a programming course. Prompted by the means established above to attain rich evidence of students' CT through their Scratch projects along with setting a stage to facilitate formative assessment in the future, we were encouraged to revise an assessment framework based on previous studies. The research questions (RQs) are as follows:

(1) What programming contents did the students' Scratch projects contain?
(2) What core educational principles in CT did the students conceptually encounter?

Methods

Research Design

This empirical study had an embedded single-case design. The studied case was an introductory programming course organized for primary school students, and the units of analysis were the Scratch projects that the students made during the course. This study intended to immerse in the particular case deeply rather than acquire data to generalize or represent the population for widespread decision-making. Previous studies have assessed students' Scratch projects in this age group and in compulsory education (e.g., Funke *et al.*, 2017). However, this study adopted novel theoretical and analytical approaches that contributed in uncovering in-depth knowledge. This knowledge, attained from one situational context, was intended for wider comparison, creation of theoretical models, stimulation of hypotheses for experimentation, and further methodological development. On that account, the employed research method was that of a descriptive case study. Description relied on the theoretical premises regarding the assessment of CT in Scratch projects, as outlined in the previous sections. The main theoretical concepts and developed framework were determined through a thorough literature review to reinforce external and code construct validity. (Yin, 2012.)

Participants

The students of three 4th grade classes from an average-sized Finnish municipal primary school participated in this study. The classes were selected because the students were surveyed as generally inexperienced at programming. Also, a prequestionnaire confirmed that the students were largely novices at programming, apart from a few previous programming experiences. The classes comprised 22, 21, and 26 students, from which 57 of them (62% girls and 38% boys) had informed consent provided by their legal guardians. The students were between 10 and 11 years old during data collection. Two of the participants were nonnative Finnish speakers. The classes also included students with special needs who participated in the programming activities but chose not to participate in data collection.

Data Collection

The data collected in this study was the Scratch projects the students programmed during a programming course that followed general guidelines in the Finnish primary school core curriculum. Each class attended the course separately one lesson per week for 4 months (13 lessons in total) in early 2017. The course was piloted with one class in another school to estimate and develop the employed pedagogical methods and data collection methods. The lessons were conducted mainly in the school's computer lab,

which had 15 functional computers. The teachers grouped the students (1-3 students per group) at the start of the course based on perceived shared skill levels or similar interest areas. The first author was the primary instructor of the course due to the regular teachers' lack of experience in programming education. The regular teacher of each class was always present, and a research assistant and learning assistant for special needs students were present during most of the lessons. All the teachers participated in guiding the students' work.

The course design was inspired by previous studies (Grover *et al.*, 2014; Meerbaum-Salant *et al.*, 2013). As the students were relatively young and new to programming, the main objective of the course was to introduce fundamental Scratch features and CT through perceivably introductory programming contents and activities to the students. The course began by discussing applications of programming in the world and "unplugged" exercises over one lesson. Subsequently, lesson-specific learning goals were targeted by programming Scratch projects (see below), which included selections from the *Creative Computing* guide (Brennan *et al.*, 2014).

The student groups programmed different kinds of projects during the course (Table 1). "Tutorial", "Debugging", and "Remix" projects involved preset objectives that guided toward designing, remixing, or debugging specific contents. Typically, these lessons began with a teacher-led demonstration of a feature (e.g., sprites sprint-racing) or an incomplete program that required implementing or error-correcting particular contents (e.g., "event-sync" code construct as the opening shot). Subsequently, the students were guided to follow the tutorial or remix and complete and creatively extend their

Table 1

Projects the students programmed during the course used as data

Project	Name	Type	Objective	Key contents	*N*
P1	"Scratch surprise"	Design	Create and modify sprites and scripts with blocks.	Scratch GUI (e.g., logging in, using blocks); experimenting	33
P2	"Cat dance"	Tutorial	Program a dance performance.	Scripting, iteration, "sequence", "event"	28
P3	"10 blocks"	Design	Plan and program your own series of instructions.	Planning, animating, "wait", "loop"	22
P4	"Debugging", part 1	Debug	Debug up to four faulty programs.	Code-reading, debugging	64
P5	"Dinosaur race"	Remix	Remix a faulty program and fix an animation.	Remixing, "initialization", "event-sync", "parallelism"	26
P6	"Riddler game"	Design	Program a game that asks questions, receives keyboard inputs and checks the correctness of answers.	"Variable", "conditional", "user interaction"	30
P7	"Debugging", part 2	Debug	Debug up to four faulty programs.	Code-reading, debugging	96
P8	Final projects	Design	Design an interactive game, story, or animation.	Planning, creative design	26
				Total:	**325**

own project. By contrast, the programmatic requirements of "Design" projects were less rigorously set: the students imagined and programmed projects within certain negotiated boundaries (e.g., a riddler game), having an opportunity to search for ideas from the Internet and, with projects P3 and P8, plan their projects with pen and paper over one lesson. All projects that the students had assigned to provided studios once the course ended were collected as data. Projects made outside the lessons were excluded because they were mainly incomplete drafts.

Data Analysis

Revising the Rubrics

As asserted previously, our priority was to aggregate manifold programming contents indicating CT thoroughly and systematically in Scratch projects. To prioritize gaining especially rich insight on the two essentially interconnected content areas, individually instantiated coding patterns and their underlying code constructs, the examinations of "other programming contents" (see Appendix C) are omitted here. We selected the PECT model's (Seiter and Foreman, 2013) voluminous rubrics as a baseline for our rubrics (described below). Several revisions to expand and regularize PECT's rubrics to patterns and constructs and convert its project-wide categorization to an instance-based one were made based on other previous studies, our initial reviews of the students' projects, and our personal experiences in Scratch as follows.

Concerning coding patterns, for instance, "Animate Motion" and "Animate Looks" were merged because sprites' all properties (e.g., position, size) are animated with the same constructs. "Conversate" was revised into "Speech and Sound" to examine separately programmed conversations using text, sound, or both. "Maintain score", which originally focused on manipulating score-like integers, was revised into a more general "Data manipulation" to also reveal manipulations of other variables, such as strings (Ericson and McKlin, 2012). Moreover, we added new ways to program the patterns, such as video/audio sensing (see Moreno-León *et al.*, 2015) and extensions (e.g., "Makey Makey") in "User interaction."

Concerning code constructs, for instance, we renamed "sequencing and looping" to "control" (Moreno-León *et al.*, 2015) and included conditional structures in it (Grover and Pea, 2018). "Parallelism" was split into parallelism "within" and "across" sprites (Meerbaum-Salant *et al.*, 2013). "Initialization" was revised to function correctly on any event if a sprite was hidden until then. "Coordination" with timing was revised to function correctly with any block with a duration. We also added new Scratch-specific constructs: "pen" (Ericson and McKlin, 2012), "I/O" (Moreno-León *et al.*, 2015), and "make-a-block" (Basu, 2019; Ota *et al.*, 2016).

Analyzing Programming Contents

In short, the analysis of programming contents in Scratch projects began from examining individually instantiated coding patterns. Each instance was analyzed in terms of

what code constructs established said instance, and this combination determined the instance's type. Each of these contents demonstrated students' conceptual encounters with CT. This analysis is next described in more detail (the analysis rubrics are presented in appendices as supplementary online material[1]).

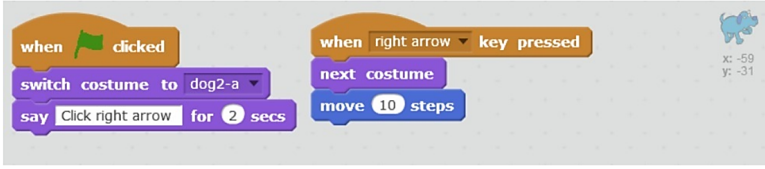
The *coding patterns* (Table 2) served as a starting point for categorizing the scripts and Scratch blocks of each sprite. Specific code constructs revealed the presence of an instance: for example, "property" code constructs revealed "Animation" pattern instances in a sprite (e.g., "show" and "hide" blocks revealed animations of visibility, see Appendix A). Similarly, "say" or "think" blocks revealed text-based "Speech and sound" instances, while "play sound" or "play note" blocks revealed sound-based ones.

Each uncovered instance was then keyed separately for all of its relevant underlying *code constructs* represented as Scratch blocks (Appendices A and B). The state of the constructs was keyed as either present (1) or missing (0) or on a 3-point nominal scale (see example in Fig. 1). Each uncovered instance in each sprite (e.g., animation of vis-

Table 2

Coding patterns in Scratch projects

Coding pattern	Instances
Animation (AN)	Modify background, costume, visibility, size, layer, an effect, facing direction, or position with timing, looping, state-sync, or event-sync
Speech and sound (SS)	Text, sound, or text-sound monologues and dialogues
Collision (CO)	Test if, repeat until, or wait until colliding with another object
Data Manipulation (DM)	Use/modify, test separately, loop until, or wait until a value in a Scratch variable, a named variable, or a named list
User Interaction (UI)	Green flag, click/key press, mouse use, keyboard input, video/audio, extensions



Sprite:	Dog
Coding pattern:	Animation (costume)
Code constructs	**State**
Property (looks)*	3 (switch costume) + 2 (next costume)
Pen	0
Initialization	1 (set costume directly on green flag)
Control	1 (sequence)
Coordination	2 (key press)
Parallelism	0
Instance type:	*Event-sync change (AN–5)*

Sprite:	Dog
Coding pattern:	User interaction (key press)
Code constructs	**State**
*UI event**	2 (key press)
*I/O**	0
Mouse	0
Control	1 (sequence)
Variable	0
Instance type:	*Click/key press (UI–1)*

Fig. 1. Code constructs and resulting types for two example coding pattern instances.

[1] See link in https://orcid.org/0000-0002-0717-5562

ibility for Dog) was considered a single instance unless the sprite's scripts comprised different "control" or "coordination" constructs for separate instances of the same type. In such cases, a new instance was detached from the original one (see example in Fig. 2). Similarly, the "Speech and Sound" instances in each sprite were considered monologues, but monologues in different sprites were merged if they established a dialogue with the "parallelism" or "coordination" constructs.

The resulting construct combinations for coding pattern instances enabled determining which particular *instance type* (e.g., "Timed animation" [AN–1], "Time-sync dialogue" [SS–2], see Appendix A) the programmed instance was if the minimum requirements for the required constructs in the instance types were met. The instance was defined as dysfunctional if the construct combinations did not meet the minimum requirements of any instance type.

The categorization resulted in a collection of different individually instantiated patterns and their underlying constructs that established the scripts in each sprite in each project. As the categorization focused on directly observable Scratch blocks following rigid rule-based coding (i.e., not requiring interpretation of the contents), it was performed by the first author by examining screenshots taken of the program code in each project using Atlas.ti software.

To analyze only relevant programming contents, the scripts in Debug and Remix projects (see Table 1), which comprised premade block segments that the students received for modification, were categorized only for segments that the students had created or changed. Tutorial, Debug, and Remix projects, which had pre-set objectives (e.g., designing specific contents), were analyzed according to how much intended content the students programmed, how the projects varied relative to that content, and what other content the projects comprised. Design projects, which were more ill-structured regarding programmatic prerequisites, were described for the content the students programmed in them.

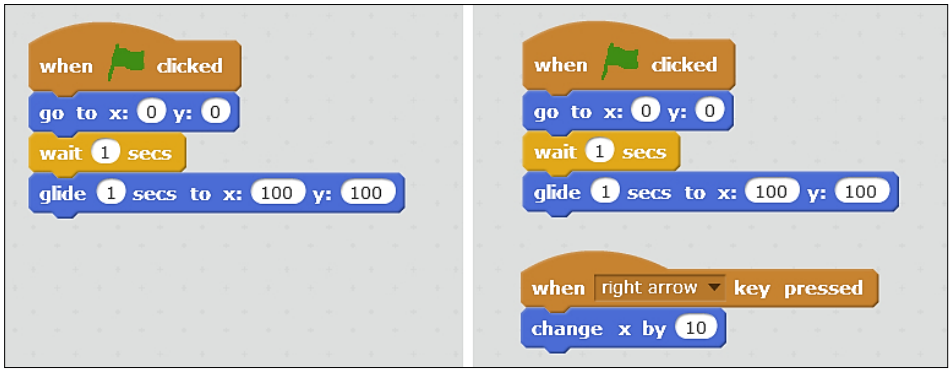


Fig. 2. Left: an instantiated "Timed animation (location)" coding pattern. Right: the location modification on the bottom (i.e., "change x by 10") is detached as a new instance because it is coordinated with another code construct, that is, "event-sync".

Interpreting Conceptual Encounters with CT

Based on our prior work in mapping Scratch programming contents and CT (Fagerlund *et al.*, 2020), conceptual encounters with the core educational principles in CT's concepts and practices (Appendix C) were logged for each student through the functional and self-designed coding pattern instances and code constructs in these instances. For example, each "Animation" coding pattern instance and each "variable (state 1)" code construct logged a conceptual encounter with the "Abstractions of properties" (Abstraction) core educational principle. Resultantly, each student's personal project portfolio (see also Brennan & Resnick, 2012), which was aggregated by the investigator from all projects the student had submitted, included a positive whole number for each CT-fostering content type.

First, the content types indicating conceptual encounters were examined if they were present in the portfolios. To determine variation in the diverse content types (e.g., the student could have implemented particular constructs more often than particular instances that both indicate an encounter with a specific principle), comparability needed to be established: coefficients of variation were computed as a quotient of mean and standard deviation for each content type.

Additionally, to provide an overview of each conceptual encounter, some of which were indicated by potentially more than one content type (e.g., "Sprites' properties", "Variables", and "Lists" all indicate an encounter with "Abstractions of properties"), the presence of each content type within each core educational principle was totaled.

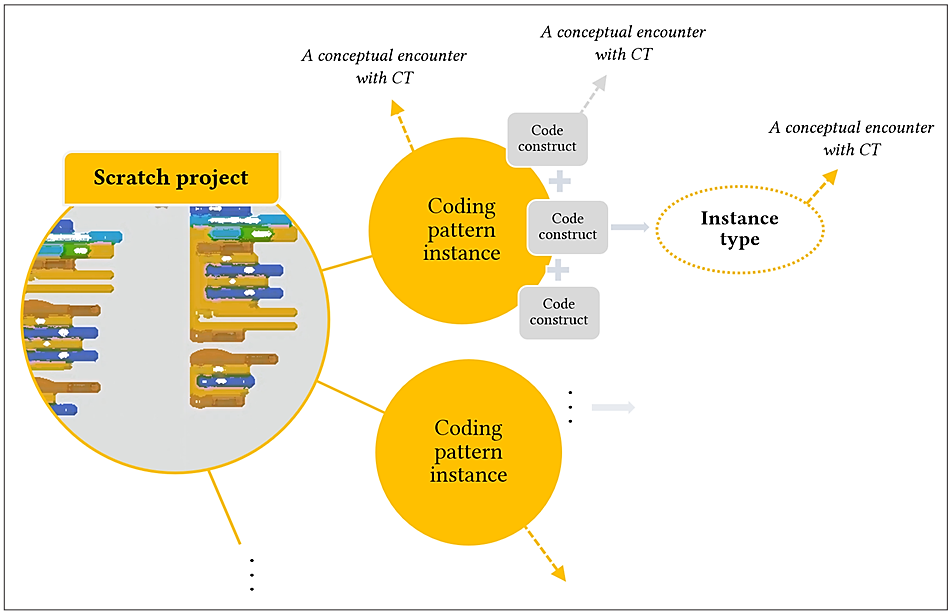


Fig. 3. Analysis of programming contents in Scratch projects in this study.

Subsequently, a mean presence (%) and average coefficient of variation (%) was computed for encounters in each core educational principles as a whole.

An overview of the analysis is portrayed in Fig. 3. In summary, any and all individually coding pattern instances (see Table 2) were examined from each Scratch project. Each instance was analyzed in terms of what relevant code constructs established it, and this combination determined the instance's predetermined type (see Appendices A and B). Each of these contents demonstrated the CT the authoring student conceptually encountered (see Appendix C).

Results

Programming Contents in Students' Scratch Projects (RQ1)

Tutorials

"Getting Started with Scratch" (P2) was a Tutorial that was accessed through the Help menu directly in the Scratch editor (see examples in Figures 4 and 5). All submitted projects ($N = 28$) comprised the instructed "Green flag" (UI–1), "Time-sync animation (location)" (AN–1) and "Sound monologue" (SS–1) instance types, and all projects but one of them contained the instructed "Text monologue" (SS–1). None of the projects comprised the remaining instructed instance types, indicating that the projects were incomplete or that contents were removed after completing the tutorial. Therefore, the final median and mode completion rates of the tutorials were 50% (four of eight instances). However, each project entailed uninstructed instances (*Mdn* = 2, *Max* = 7) in the "Animation" (total: 24), "Speech and sound" (5), and "User interaction" (1) patterns, indicating that the students had proceeded to custom design halfway through the tutorial or after having removed contents from the finished tutorial.

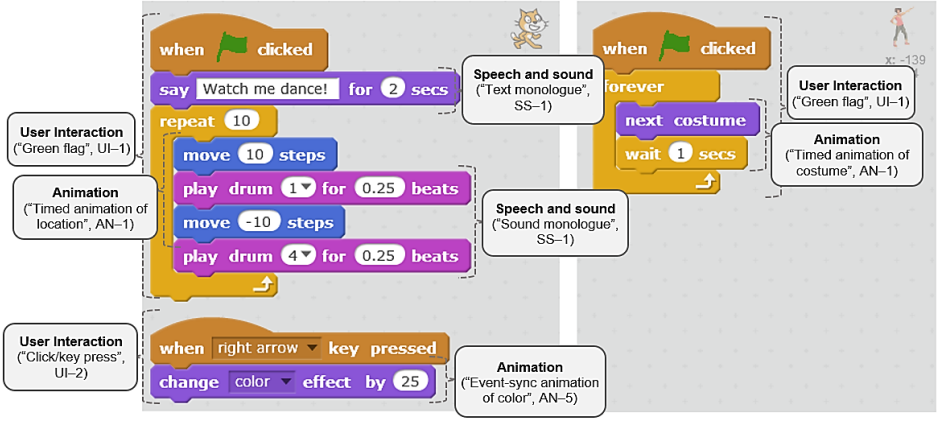


Fig. 4. A fully completed "Getting Started with Scratch" tutorial and the instances as instructed by it.

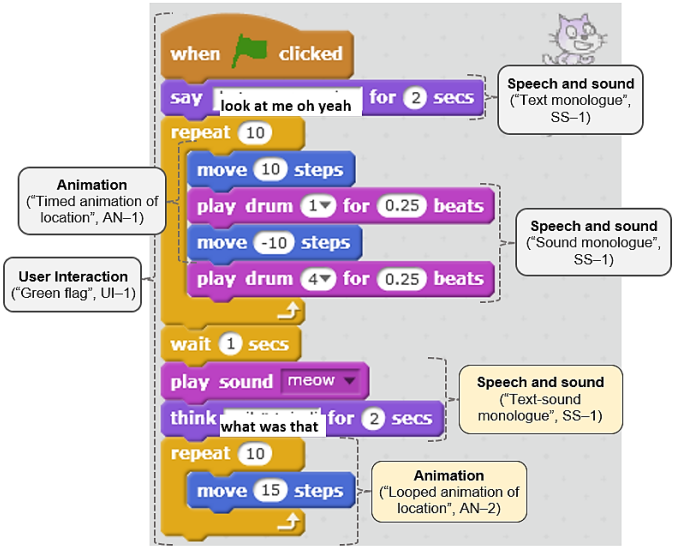


Fig. 5. A half-completed tutorial including two uninstructed instances.

Remixes

In the P5 projects ($N = 26$), the students were tasked to remix an incomplete project and add the "Event-sync animation (location)" (AN–5) instance type with the "initialization" code construct for two separate sprites. Twenty-two (85%) of these projects met these requirements whereas the remaining four projects (15%) comprised the "event-sync" construct for starting scripts that entailed location animations, but the locations were not initialized (see comparison in Fig. 6).

Similar to the P2 (Tutorial), 69% of the projects comprised other instance types than those that the students were minimally required to implement ($Mdn = 3$, $Max = 9$) again exclusively in the "Animation" (total: 39), "Speech and sound" (24), and "User Interaction" (3) patterns.

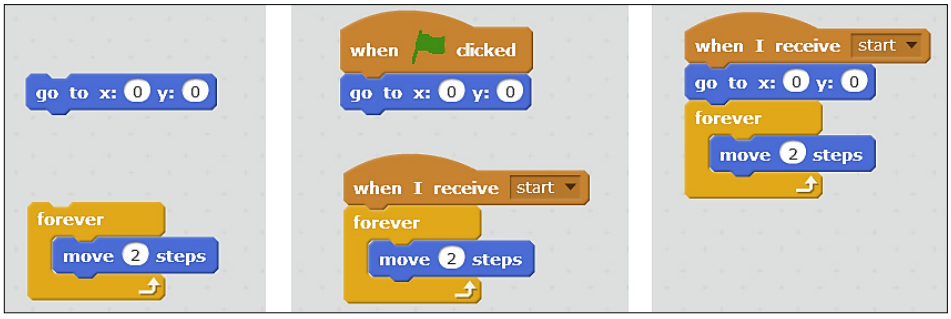


Fig. 6. Left: the initial problem. Center: "Event-sync animation (location)" (AN–5) with "initialization." Right: "Event-sync animation (location)" (AN–5) followed by "Looped animation (location)" (AN–2) with no "initialization".

Debugging Challenges

Debugging challenges parts one (P4) and two (P7) each comprised four faulty projects, which the students were guided to begin correcting, potentially submitting all four of them. In contrast to P7, the numbers of submitted projects decreased drastically in P4 (Table 3), indicating that many students struggled with P4.3 (looped animation of location with move and bounce).

In P4.1, the students were challenged to implement "initialization" to preexisting animation of location. Most groups submitted incorrect responses, for example, by programming the sprite to glide back to its starting location prior to program termination. In P4.2, although all submitted projects comprised a programmatically functional instantiated "Animation (direction)" pattern, complete evaluation of correctness required manually verifying that the sprite rotated 360 degrees. In P4.3, the two incorrect responses lacked the "bounce" code construct.

In P7.3 and P7.4, the challenge was to change the parameters in preexisting "Test collision in loop" (CO–2) and "Loop until costume #" (DM–3) instance types. Manual observation was again required to verify the correctness of the parameters. Only one response in each project respectively was incorrect, comprising "repeat" blocks instead of conditional looping.

Design Projects

The first type of Design project that the students programmed during the course was themed as "Riddler games" (P6, $N = 30$). In these projects, the students were instructed to design a game that asks questions, receives keyboard inputs as responses, and evaluates the correctness of the answers. Programmatically, the game minimally required a "Keyboard input" (UI–4) and an appropriate instance type in "Data Manipulation" to test a stored value in the "answer" variable (i.e., DM–2, DM–3, or DM–4). All but two projects (93%) comprised both instances. These two projects involved "ask" blocks

Table 3
P4 and P7 debugging projects solved by student groups

Project	Debugging objective	Submitted		
		N	Correct	Solved
P4				
4.1	"Timed animation (location)" (AN–1) with "initialization"	27	19%	19%
4.2	"Animation (direction)" (any instance type) with 360° rotation(*)	25	100%	93%
4.3	"Looped animation (location)" (AN–2) with "move" and "bounce"	10	80%	30%
4.4	"Text–sound monologue" (SS–1)	3	100%	11%
P7				
7.1	"Event-sync animation (costume)" (AN–5) with "repeat"	27	96%	96%
7.2	"Event-sync animation (stop)" (AN–5) in four different sprites	25	100%	93%
7.3	"Test collision in loop" (CO–2) with "Nano" parameter(*)	23	96%	81%
7.4	"Loop until (costume #)" (DM–3) with correct condition to finish looping(*)	21	95%	74%

*The rubrics themselves did not verify the use of correct parameters.

for question-asking and "if-else" blocks for answer checking, but these blocks were unscripted, rendering them dysfunctional and indicating that the projects were unfinished. All functional answer tests were conditional structures in "Test value" (DM–2). In addition to the instructed requirements, all the projects entailed other instance types (*Mdn* = 5, *Max* = 11) exclusively in the "Animation" (total: 55), "Speech and sound" (62), and "User Interaction" (49) patterns.

The students had more creative freedom for the other Design projects. These projects included "Scratch Surprise" (P1), the students' first self-designed Scratch project; "10 Blocks" (P3) as exercises in script planning; and interactive games, stories, or animations (P8) as final project assignments. The division of functional instances in these projects (Table 4) revealed that "Animation" was substantially the most commonly instantiated pattern (49% of all instances), followed by "User Interaction" (25%) and "Speech and sound" (20%). "Data Manipulation" (4%) and "Collision" (2%) were rarely instantiated.

The P1 projects ($N = 33$) typically comprised various blocks as nascent scripts but without events to start them (see examples in Fig. 7). As a result, 58% of all instances were dysfunctional, suggesting that the students did not spontaneously grasp event-driven scripting entirely or merely experimented with different features. The most common functional instance types were "Monologue" (SS–1) (total: 28, in 45% of the projects), "Green flag" (UI–1) (total: 22, in 42%), and "Timed animation" (total: 21, in 30%).

Table 4

The numbers of instantiated coding patterns in the three open-ended design projects that students programmed during the course

Coding pattern	Instantiated in open-ended design projects			Total
	P1 "Scratch surprise" ($N = 33$)	P3 "10 blocks" ($N = 22$)	P8 Final projects ($N = 26$)	
Animation	83	77	417	577
Speech and sound	64	50	127	241
Collision	3	0	36	39
Data Manipulation	17	0	40	57
User Interaction	30	27	200	257

Fig. 7. Sample unscripted blocks from three P1 projects.

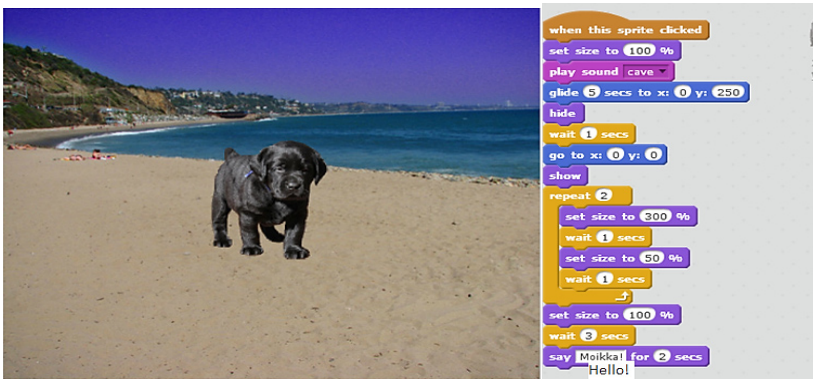


Fig. 8. An example P3 project.

The P3 projects ($N = 22$) had typically one or two sprites performing simple behaviors, such as introducing themselves with "Timed animation" (AN–1) (total: 45, in 86% of the projects), "Monologue" (SS–1) (total: 39, in 86%), and "Click/Key press" (UI–1) (total: 15, in 59%) (see Fig. 8). However, in contrast to the P1 projects, only 12% of all instances in these projects were dysfunctional, indicating that the students had begun internalizing the idea behind scripting.

The final project assignments, the P8 projects ($N = 26$), entailed thematically and programmatically versatile game, animation and story-like projects (see Fig. 9 and

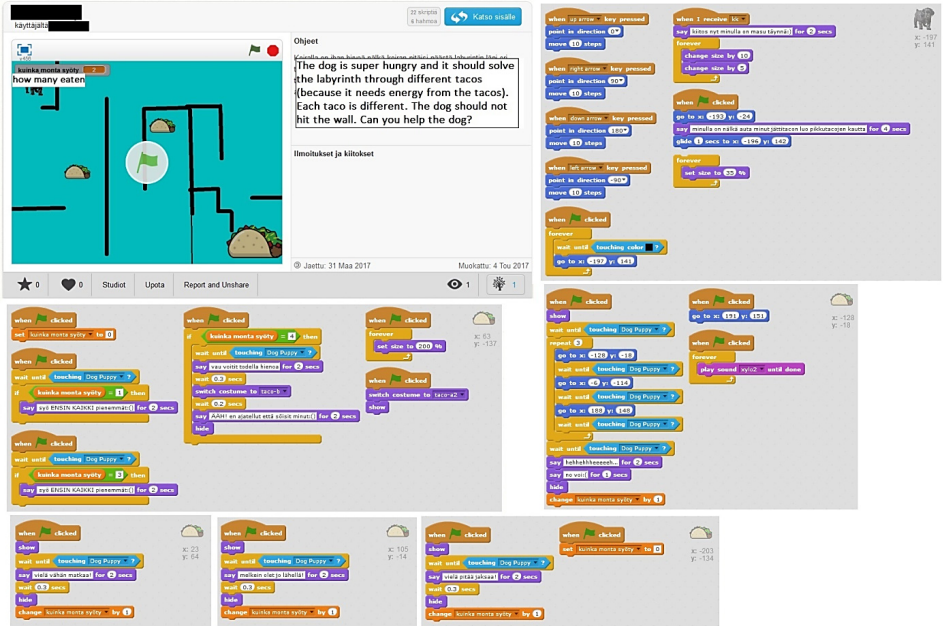


Fig. 9. The components and scripts in a relatively complex P8 project.

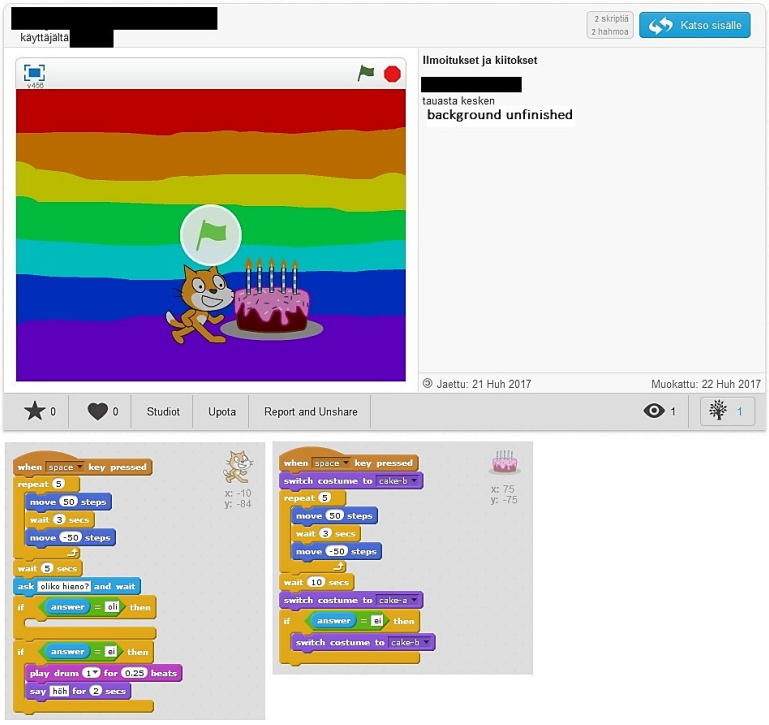


Fig. 10. The components and scripts in a relatively simple P8 project.

Fig. 10). 9% of the instances in these projects were dysfunctional, suggesting that the students still struggled with implementing contents or that the projects were not entirely finished. Of the functional instance types, the most common were "Event-sync animation" (AN–5) (total: 258, present in 89% of the projects), "Green flag" (UI–1) (total: 113, in 92%), and "Monologue" (SS–1) (total: 57, present in 81% of the projects).

Project Portfolios

The students' portfolios (N = 57) contained between 2 to 14 projects (*Mdn* = 10), suggesting that few students participated in designing only two projects or that all portfolios did not include all programmed projects. Nevertheless, the portfolios demonstrated the varying numbers of instance types that the students programmed during the course (Table 5). The most common types were "Event-sync animation" (AN–5), "Monologue" (SS–1), and "Green flag" (UI–1). More than half of the instance types received a median of zero, potentially highlighting more advanced contents.

Similar statistics were computable for the code constructs as well, but their high number rendered reporting inappropriate; however, the most common constructs were "sequence" (*Mdn* = 55), "repeat"/"forever" (*Mdn* = 22), and "green flag" (*Mdn* = 16).

Table 5
Minimum, maximum, and median numbers of instance types in students' project portfolios

Instance types of coding patterns	Numbers in project portfolios		
	Min	Max	Median
Animation (AN)			
1–Timed animation	0	32	4
2–Looped animation	0	20	6
3–State-sync animation (repeat until)	0	11	0
4–State-sync animation (wait until)	0	11	0
5–Event-sync animation	0	146	9
Speech and sound (SS)			
1–Monologue	2	18	7
2–Time-sync dialogue	0	6	0
3–State-sync (repeat until) dialogue	0	0	–
4–State-sync (wait until) dialogue	0	7	0
5–Event-sync dialogue	0	15	0
Collision (CO)			
1–Test collision separately	0	5	0
2–Test collision in loop	0	5	1
3–Wait for collision	0	3	0
Data manipulation (DM)			
1–Use/modify variable	0	6	0
2–Test value	0	5	1
3–Loop until value	0	2	1
4–Wait for value	0	0	–
User interaction (UI)			
1–Green flag	1	25	6
2–Click/key press	0	20	2
3–Mouse use	0	4	0
4–Keyboard input	0	5	1
5–Video/audio	0	0	–
6–Extensions	0	0	–

Missing Code constructs

Examining code constructs in the instances revealed that the P1 projects were often missing the "control" and "coordination" constructs (Table 6), which essentially rendered most instances dysfunctional. Although the lack of these constructs decreased greatly in subsequent projects, they were still occasionally missing, indicating recurring difficulties or unfinished projects. "Initialization" remained as a frequently missing construct throughout the course.

Students' Conceptual Encounters in CT (RQ2)

According to the students' conceptual encounters in CT, as indicated by the programming contents in their project portfolios (Table 7), all the students re-instantiated the coding patterns and code constructs (Patterns) and decomposed the projects into smaller

Table 6

Code constructs missing from coding pattern instances in the students' projects

Code construct	Missing from projects							
	P1	P2	P3	P4	P5	P6	P7	P8
"Control"								
Number	99	8	7	4	2	15	2	30
Percentage	50%	5%	5%	4%	2%	7%	1%	4%
"Coordination"								
Number	113	15	16	6	3	24	6	54
Percentage	57%	10%	10%	7%	3%	11%	3%	7%
"Initialization"(*)								
Number	24	41	61	73	37	33	587	131
Percentage	83%	100%	91%	92%	45%	73%	100%	35%

*Only in functional Animation instances.

Table 7

Presence of and variation among conceptual encounters with CT's core educational principles in students' project portfolios

CT concept/practice	Core educational principle	Indication in project portfolios	
		Presence	Coefficient of variation
Abstraction	Abstractions of behaviors	34%	277.4
	Abstractions of properties	63%	66.3
	Abstractions of states	54%	291.2
Algorithms	Algorithm control	91%	84.6
	Procedures	34%	277.4
	Starting from initial state	93%	216.6
	Recursion	0%	–
Automation	I/O devices	33%	152.1
Coordination	Coordinating scripts	57%	183.8
	Synchronizing scripts	18%	437.5
Creativity	Modifying remixes	81%	69.8
Data	Storing and manipulating data	50%	240.6
Logic	Boolean logic	0%	–
	Conditional structures	44%	239.0
	Operations	96%	104.5
Modeling and design	Algorithm animation	66%	58.6
Patterns	Re-instantiated coding patterns/code constructs	100%	23.1
Problem decomposition	Decomposition	100%	76.3
	Modularized features	63%	153.9

parts (Problem decomposition). Nearly all (>90%) the students designed complex projects (Abstraction), implemented algorithm control structures and "initialization" (Algorithms), remixed (Collaboration), and utilized logical operators (Logic).

However, less than half (<50%) of the students abstracted behaviors for sprites (Abstraction), used procedures (Algorithms), utilized I/O devices (Automation), and synchronized parallel scripts (Coordination). None of the students implemented recursive solutions (Algorithms) or Boolean logic (Logic). Variation was large among instantiating synchronized parallel scripts (Coordination), demonstrating that the few students who encountered this principle did so several times.

Discussion

CT through programming is a new topic in primary education that necessitates evidence-based pedagogical knowledge, especially regarding assessment that enhances learning (Lye and Koh, 2014). This study assessed 4th grade students' CT by focusing on their Scratch projects designed in naturalistic classroom situations. We adopted a comparatively inclusive view of what students can learn in CT through Scratch and, by revising a profound assessment framework, focused uniquely on individually instantiated coding patterns and their underlying code constructs, that is, relatively fine-grained evidence. The framework uncovered ample and manifold empirical findings of contents programmed by the students and respective indications of their conceptual encounters with CT. Next, we discuss the significance that this evidence and employing the assessment framework may have in teaching and learning CT in Scratch, highlighting also limitations that our analysis poses. Moreover, we address our outlying goal: developing formative assessment systems in schools.

Programming Contents Indicating CT

Coding Patterns

The students implemented instances of "Animation", "Speech and Sound", and "User Interaction" by far the most, specifying previous findings (Seiter and Foreman, 2013) concerning that these patterns are altogether most typically present in students' projects. These patterns were also exclusively volitionally designed. Concerning conceptual encounters with CT, these contents indicated that the students repeatedly experienced abstracting properties and behaviors (Abstraction), designing procedures (Algorithms), animating algorithms (Modeling and design), manipulating pre-provided data (Data), decomposing projects into coding patterns and code constructs (Problem decomposition), and reinstantiating patterns and constructs (Patterns). These experiences could be expected to occur somewhat naturalistically in Scratch, which is essentially a tool for designing interactive media (Brennan and Resnick, 2012).

Perhaps more intriguing and relevant for pedagogical consideration is that, by contrast, "Data manipulation" and "Collision" were seldom designed. This influenced the students' conceptual encounters with CT mainly via the relative scarcity of variables, conditionals, and logical operations (specified in the following sections), which can, however, be considered as fairly fundamental computational concepts (Grover and Pea, 2018). An

underlying cause may concern the designed types of projects: Moreno-León *et al.* (2017) showed that the presence of certain constructs typically varies between projects in different genres. We perceived the students' projects most akin to animations and stories with little interactivity. Therefore, they may have lacked opportunities to explore supplementary genres, such as simulations or more sophisticated games to which Data Manipulation and Collision may be more typical (see Seiter and Foreman, 2013). Facilitating the design of such presumably more complex projects can be justified in advanced stages of learning CT. These patterns were also not systematically introduced during the course, proposing that students may be inclined to volitionally designing familiar contents and that they could benefit from deliberate guidance towards unfamiliar contents.

Instance Types

Our systematic categorization of instance types in the coding patterns allowed analyzing the students' CT in novel detail. The most often instantiated instance types, such as "event-sync animation" (AN–5), "monologue" (SS–1), and "green flag" (UI–1) (see Table 5), indicated that the students repeatedly experienced coordinating scripts with timing and events (Coordination), controlling algorithms by sequencing and looping (Algorithms), and modularizing animations and speaker roles (Problem decomposition). The prominence of these experiences may stem from the nature of event-driven programming in Scratch (Maloney al., 2010) and blocks representing code constructs that novice programmers typically first learn to use (see Grover *et al.*, 2014).

Again, perhaps more interesting and allusive in terms of CT pedagogy was that the students sporadically experienced utilizing conditional logic and arithmetic operations (Logic), abstracting program states with continuous events (Abstraction), coordinating scripts with states (Coordination), modularizing data manipulation and collision detection (Problem decomposition), and utilizing key pressing, clicking, and keyboard inputs (Automation). Supplementing prior studies (Burke, 2012; Franklin *et al.*, 2013; Maloney *et al.*, 2008), these findings specified exactly how students implement user interaction in their projects: in this study, they mainly implemented "green flag" instead of different I/O devices, indicating that the projects typically lacked usability and, consequently, resembled projects more for viewing than playing. As discussed above, the other features, such as logical operations and collision detection, may be more typical to presumably more advanced game-like projects (Moreno-León *et al.*, 2017), proposing a need for educators to purposefully introduce game-like features in Scratch and thus CT more extensively. Despite few examples in prior studies (e.g., Burke, 2012; Sáez-López *et al.*, 2016), how the substance of different curricular topics could be processed while creatively designing usable Scratch projects, such as games and simulations, is not extensively known. The integration of CT in Scratch thoroughly across the curriculum at the primary school level presents a fruitful opportunity for pedagogical planning and further research.

Several instance types were instantiated infrequently or never. The students therefore experienced little if any abstracting program states with discrete events (Abstraction), utilizing Boolean logic (Logic), modularizing behaviors with state-sync (Problem decomposition), and utilizing mouse, video/audio, and extensions (Automation). Devices,

such as microphones or extensions like Makey Makey, which could have promoted exploring these areas in CT, were not available in the school. As the use of various I/O devices is key in CT, schools could acquire such physical add-ons for learning purposes, and their meaningful use in cross-curricular programming with Scratch could also be diversely considered. Implementing mouse use (UI–3) could also again be more typical to game-like projects, although the students may have also disregarded it because it was not explicitly taught or demonstrated. For the abundance of creative opportunities in Scratch, students could be guided to browse existing projects in the Scratch repository to gain ideas and knowledge of what possibilities exist altogether. In turn, code blocks, such as the "wait until" and Boolean operations, which essentially relate to the other above-mentioned less encountered areas of CT, are specified below.

Code Constructs

Several previous studies have examined students' use of code constructs. However, assessing them within instantianted coding patterns allowed us to gain insight regarding their use in diverse creative circumstances that were, perhaps most importantly, semantically meaningful, thus also favoring the legitimacy of the examination. Among notable findings was that the students' first projects (P1) comprised mainly unscripted blocks and parameter state changes without "coordination", suggesting that the students did not intrinsically grasp controlling algorithms (Algorithms) and coordinating them with, for instance, timing (Coordination). Control and coordination became greatly more prevalent after the students had completed the scripting tutorial (see Table 6), suggesting that direct instruction can be effective for learning these fundamentals of programming and an effective way to launch especially introductory courses in schools. However, these constructs were occasionally still missing in the final projects (P8), possibly exhibiting "bad programming habits" (Moreno-León *et al.*, 2015), situated here under other programming contents (see Appendix C), and highlighting a need to remind students to maintain their use. However, the projects may have been incomplete, suggesting a lack of time and underlining the ever-challenging need for educators to ensure sufficient time for designing.

The students typically controlled the programmed instances with "sequences" and "loops" and coordinated them with "timing" and "event-sync" (see Table 5). "Conditional looping" (i.e., the "repeat until" block) and "conditional structures" were rare in control whereas "state-sync", "blocking", and "stopping" were rare in coordination. Consequently, the students seldom encountered the different ways to control algorithms (Algorithms) and coordinate automated processes (Coordination). Coordination by stopping and blocking may be somewhat exceptional in Scratch: stopping causes repeating animations to halt, being relevant mainly in projects involving infinite looping in "looped animation" (AN–2) (e.g., stopping a sprite from moving forever), and blocking is established with the "ask/set and wait" block, being relevant mainly in projects where "keyboard input" (UI–4) blocks the execution of an "Animation" pattern (e.g., sprite motion temporarily stopped to receive a specific input). However, because these contents are relevant for CT, an opportunity remains to consider pedagogically meaningful ways to incorporate them in programming tasks.

Then again, "repeat until" and "wait until", which represent "state-sync", are blocks that have been noted to be difficult for students to use (Basu, 2019; Seiter and Foreman, 2013). These blocks can be used, for instance, to synchronize collision detection (e.g., CO–2, CO–3) and speaker roles in dialogues (e.g., AN–3, AN–4), highlighting game-like and story-like projects with colliding and conversing sprites as potentially meaningful – although presumably more advanced – contexts to introduce these constructs. However, most students debugged the "repeat until" block correctly in a structured debugging challenge (P7.4), suggesting that such challenges could offer a viable route between direct instruction and more open-ended design to teach students to understand and use even more advanced constructs.

Concurring with the findings of Franklin *et al.* (2013, 2017), "initialization" was missing to varying degrees throughout the course. Most students debugged initialization correctly in P4, however, few students demonstrated avoiding it by programming the sprite to glide back to the starting location. Initialization was explicitly instructed with P5, which may have reflected on its high presence (see Table 6). Nevertheless, we found that its presence subsequently decreased and varied, suggesting that a conceptual encounter may not self-evidently guarantee gaining a deep understanding. Instead, encountering contents repeatedly over time can be necessary for enhancing understanding and developing more rigorous skills. However, initialization is not mandatory for programs to execute in Scratch, contesting whether Scratch facilitates conceptually encountering it consistently and demonstrating students' understanding reliably in it.

The students manipulated exclusively Scratch variables, resulting in no experiences with abstracting properties as custom variables and lists and manipulating them (Abstraction, Data). Similarly, the lack of arithmetic and Boolean operations revealed that the core educational principles in Logic remained largely unencountered. Moreover, the students rarely implemented "parallelism", resulting in sparse experiences in synchronization (Coordination). Variables, Boolean operations, and parallelism have been previously discovered to be somewhat difficult for students to understand and use (Basu, 2019; Maloney *et al.*, 2008; Meerbaum-Salant *et al.*, 2013; Seiter and Foreman, 2013). In Scratch, parallelism could be introduced meaningfully when synchronizing animations (e.g., AN–4), establishing dialogues (e.g., SS–2), or waiting for sprites to collide (e.g., CO-3) especially in game-like projects. For variables and logical operations, students could design Data Manipulation with comparisons (DM–2, DM–3, or DM–4) in, for instance, a math quiz project.

Lastly, the students never used "pen", which can visualize sprites' movement paths (Ericson and McKlin, 2012) and, therefore, animate algorithms (Modeling and design). However, algorithm animation occurs naturalistically through most programmed features in Scratch, contesting the significance of this construct. Moreover, "make-a-blocks" were nonexistent, and only one student used "cloning" once. Consequently, the students mainly never abstracted and programmed custom behaviors or clones' behaviors (Abstraction, Algorithms) or implemented recursive solutions (Algorithms). These constructs have been rarely addressed in prior K–9 studies, suggesting that they, in addition to other contents that were rarely implemented, may better suit more experienced programmers.

Implications for Research and Practice

Despite the prevalent ideology of interest-driven design and discovery-based learning in Scratch (Brennan and Resnick, 2012), direct instruction and structured debugging could effectively introduce students to fundamental contents and contents which they cannot manage to implement or fail to realize as hidden possibilities. These approaches can be purposeful when students begin to learn scripting in Scratch, become later introduced with such fundamental constructs as "initialization", and are guided to realize previously unknown creative opportunities, such as mouse use (UI–3). Investigating how the instruction of particular contents (e.g., user interaction) could pave way for students' constructive less-structured explorations (e.g., moving from green flag to other kinds of interactivity) and how students' interactions with various resources in different tasks could lead to successful content implementations could be pedagogically informative. The model of scope of autonomy recently introduced by Carlborg *et al.* (2019) could provide a vignette through which to examine such issues. Moreover, our results suggest that a mere conceptual encounter may not assure gaining robust knowledge, and learning through implementing contents requires repetitions. Therefore, we restate known concerns (e.g., Lye & Koh, 2014) providing reason to meticulously examine when and how students gain genuine skills and deep understanding in CT while programming.

For learning CT comprehensively, it can be important to design various, presumably more complex kinds of projects, including narratives with several speakers, games with colliding objects and score count, projects with data manipulation, and, altogether, projects that are usable with different I/O devices. Creative contexts in which students could implement such contents, especially the seemingly more advanced ones (e.g., Data Manipulation, Collision, coordination by stopping and blocking, custom variables, Boolean operations), could be adapted from the rubrics in future empirical studies. This could be to examine their feasibility along with considering how to organize compact yet fruitful programming courses in schools. It seems especially important for practitioners to find time to introduce the potentially more complex contents through more complex projects (e.g., games and simulations). The rubrics employed herein may suggest some content organization and the results may suggest the kinds of programming capabilities that students may gain more intrinsically than they do others. However, developing rigid learning trajectories applying, for instance, the Bloom/SOLO taxonomy (e.g., Meerbaum-Salant *et al.*, 2013) for contents would require more studies. In practice, however, it is pedagogically justifiable to offer a "high ceiling" for students to potentially reach (Brennan and Resnick, 2012).

Limitations in Analysis

Although programmed artefacts are latent manifestations of thinking, evidence to reinforce their validity in analyzing CT has begun to emerge. For instance, analysis by Dr.

Scratch, whose rubrics were included in our framework, has been convergent with educators' grades, various software complexity metrics, and CT tests (Román-González *et al.*, 2019). We aimed to reinforce validity by building our rubrics on existing frameworks and, especially, focusing on semantically meaningful contents that the students had assuredly encountered. Nevertheless, finished projects may not have exposed all relevant evidence especially gained by ways that deviate excessively from implementing contents (e.g., code-reading, social interactions). Hence, it is vital to complement assessment with other methods, such as examining students' programming processes (Basso *et al.*, 2018; Grover *et al.*, 2017).

The students' conceptual encounters with CT were problematic to analyze deeply regarding the quality of gained skills and understanding. For instance, systematically investigating learning progressions would have required examining more projects. Additionally, this study did not investigate other programming contents, such as "no extraneous blocks" (see Appendix C), which could have complemented the findings.

The rubrics covered most blocks available in Scratch 2.0, allowing the assumption that they are relatively comprehensive. However, parametric precision (see Meerbaum-Salant *et al.*, 2013) was not analyzed as it would have required labor-intensive interpreting of sprites' parameters in different program states. Moreover, large projects may include more complex contents, such as synchronized coding patterns (see Seiter, 2015), which we similarly determined too labor-intensive to categorize. CT also embodies aspects that were difficult to instrumentalize in Scratch, such as recognizing computing in the world (Barr and Stephenson, 2011; Csizmadia *et al.*, 2015). Therefore, the rubrics should be interpreted as representing core CT-fostering contents and not necessarily as all-inclusive.

Approaching Formative Assessment

Learning goals for CT represented as programming contents can be presented relatively comprehensively to students with the rubrics in Appendix C. Educators could systematically introduce CT in semantically meaningful contexts through storytelling, animating, or game development in more open-ended or structured programming tasks that are thematically connected to different curricular areas (Bocconi *et al.*, 2018). As exemplified in this work, eliciting evidence of students' skills and understanding in CT can be carried out by assessing students' Scratch projects. To moderate the hindrance concerning slow and laborious manual analysis, assessment could focus only on selected code segments.

The purpose of feedback is to stimulate the correction of specific errors or poor strategies with clear suggestions on how to improve the work based on progress toward achieving goals (Black and Wiliam, 1998). Feedback in micro-programmatic analysis can, firstly, pinpoint errors directly or by hinting when fundamental constructs (e.g., control, coordination, initialization) are missing from a specific instance. Second, more generally, it can guide towards improving the current instance types (e.g., relative in-

stead of absolute parameter changes, synchronized dialogues instead of monologues, using different synchronization methods) or provide tutorials demonstrating how to design new instance types (e.g., score counting as Data Manipulation, Collision of sprites in a game).

Altogether, the formative assessment processes discussed above can be carried out by the teacher. However, as highlighted in formative assessment, especially for nurturing students' metacognition and collaboration (Black and Wiliam, 2009), students could assess their own or their peers' projects. Our aspiration is that the rubrics could be automated to be employed in a learning-support system that can assess projects accurately and provide timely suggestions (see also Moreno-León *et al.*, 2015).

Conclusions

CT continues finding foothold through programming in schools, although it has been enveloped by a scarcity of research focused especially on supporting learning. Pushing from such circumstance, this study used a comparatively comprehensive and fine-grained framework aimed towards enhancing especially primary school students' learning of CT. We assessed the programming contents and indicative conceptual encounters with CT through 4th grade students' versatile Scratch projects. The results provided in-depth insight of students' experiences with diverse areas in CT and the future steps of assessing it in Scratch in classroom situations.

To target the acquisition of CT through Scratch broadly in the classroom, it can be necessary to introduce manifold programming activities and design various kinds of projects apart from merely those that are especially characteristic to the tool. Pedagogical focus could be placed especially on guiding students towards unfamiliar and more advanced contents and creative possibilities. However, returning to familiar contents may be necessary occasionally to reinforce skills. Direct instruction and structured debugging can accompany the prevalent discovery-based learning approaches. Rather worryingly, however, available time to complete very intricate projects during lessons can be limited, which accentuates the potential benefit of incorporating programming in different curricular areas. Programming courses could thus promote designing usable projects that gamify or simulate other curricular topics. Devising and testing such learning tasks in practice provides an important aspiration for pedagogical planning and further investigation. However, Scratch can be effective for acquiring certain areas in CT, which should be learned in various contexts.

Concerning holistic assessment of CT, this study presents a framework for assessing particular areas in CT through Scratch projects. Future scholarly works could include large-scale reports of CT encountered over periods of time (e.g., entire curricula) and detailed investigations into individual students' experiences. Future studies could especially examine how the rubrics could be used to support learning in dynamic classroom contexts in the ways theorized herein. However, it is altogether important to complement the assessment of static projects with other methods.

Acknowledgements

We gratefully acknowledge the Department of Teacher Education at the University of Jyväskylä for facilitation of this study. Special thanks to the Innokas Network for the practical insights.

Funding details

This work was supported by the Department of Teacher Education in the University of Jyväskylä, the Central Finland Regional Fund under Grant 30161702, the Emil Aaltonen Foundation under Grant 170028 N1, and the Ellen and Artturi Nyyssönen Foundation.

References

Angeli, C., Voogt, J., Fluck, A., Webb, M., Cox, M., Malyn-Smith, J., Zagami, J. (2016). A K–6 computational thinking curriculum framework: Implications for teacher knowledge. *Educational Technology and Society*, 19(3), 47–57.

Barr, V., Stephenson, C. (2011). Bringing computational thinking to K–12: What is involved and what is the role of the computer science education community? *ACM Inroads*, 2(1), 48–54.

Basso, D., Fronza, I., Colombi, A., Pahl, C. (2018) Improving Assessment of Computational Thinking Through a Comprehensive Framework. In: Joy, M., Ihantola, P. (Eds.), *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18)* (Article No. 15). New York, NY: ACM.

Basu, S. (2019). Using Rubrics Integrating Design and Coding to Assess Middle School Students' Open-ended Block-based Programming Projects. In: Hawthorne, E. K., Pérez-Quiñones, M.A. (Eds.), *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)* (pp. 1211–1217). New York, NY: ACM.

Black, D., Wiliam, D. (1998). Assessment and classroom learning. *Assessment in Education: Principles, Policy and Practice*, 5(1), 7–74.

Black, D., Wiliam, D. (2009). Developing the theory of formative assessment. *Educational Assessment, Evaluation and Accountability*, 21(1), 5–31.

Bocconi, S., Chioccariello, A., Earp, J. (2018). *The Nordic Approach to Introducing Computational Thinking and Programming in Compulsory Education*. Report prepared for the Nordic@BETT2018 Steering Group.

Brennan, K., Balch, C., Chung, M. (2014). *Creative computing.* Cambridge, MA: Harvard Graduate School of Education.

Brennan, K., Resnick, M. (2012). *New frameworks for studying and assessing the development of computational thinking.* Paper presented at the meeting of AERA 2012, Vancouver, BC.

Burke, Q. (2012). The markings of a new pencil: Introducing programming-as-writing in the middle school classroom. *Journal of Media Literacy Education*, 4(2), 121–135.

Carlborg, N., Tyrén, M., Heath, C., Eriksson, E. (2019). The scope of autonomy when teaching computational thinking in primary school. *International Journal of Child-Computer Interaction*, 21, 130–139.

Csizmadia, A., Curzon, P., Dorling, M., Humphreys, S., Ng, T., Selby, C., Woollard, J. (2015). *Computational thinking - A guide for teachers*. `https://community.computingatschool.org.uk/files/6695/original.pdf`

Denning, P., & Tedre, M. (2019). *Computational Thinking.* MIT Press Ltd.

Ericson, B., McKlin, T. (2012). Effective and sustainable computing summer camps. . In: Smith King, L., Musicant, D.R. (Eds.), *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)* (pp. 289–294). New York, NY: ACM.

Fagerlund, J., Häkkinen, P., Vesisenaho, M., Viiri, J. (2020). Computational Thinking in Programming with Scratch in Primary Schools: A Systematic Review. *Computer Applications in Engineering Education*, 1– 17. https://doi.org/10.1002/cae.22255

Franklin, D., Conrad, P., Boe, B., Nilsen, K., Hill, C., Len, M., . . . Waite, R. (2013). Assessment of computer science learning in a Scratch-based outreach program. In: Camp, T., Tymann, P. (Eds.), *Proceedings of the 44th ACM technical symposium on Computer science education (SIGCSE '13)* (pp. 371–376). New York, NY: ACM

Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., . . . Harlow, D. (2017). Using upper-elementary student performance to understand conceptual sequencing in a blocks-based curriculum. In: Caspersen, M.E., Edwards, S.H. (Eds.), *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)* (pp. 231–236). New York, NY: ACM.

Funke, A., Geldreich, K., Hubwieser, P. (2017). *Analysis of scratch projects of an introductory programming course for primary school students.* Paper presented at the 2017 IEEE Global Engineering Education Conference, Athens, Greece.

Garneli, B., Giannakos, M., Chorianopoulos, K. (2015). *Computing Education in K–12 Schools. A Review of the Literature.* Paper presented at the 2015 IEEE Global Engineering Education Conference, Tallinn, Estonia.

Grover, S., Bienkowski, M., Basu, S., Eagle, M., Diana, N., Stamper, J. (2017). A framework for hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming. In: Wise, A., Winne, P.H., Lynch, G. (Eds.), *Proceedings of the Seventh International Learning Analytics & Knowledge Conference (LAK '17)* (pp. 530–531). New York, NY: ACM.

Grover, S., Cooper, S., Pea, R. (2014). Assessing computational learning in K-12. In: Cajander, Å., Daniels, M. (Eds.), *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education* (pp. 57–62). New York, NY: ACM.

Grover, S., Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher*, 42(1), 38–43.

Grover, S., Pea, R. (2018). Computational thinking: A competency whose time has come. In: Sentance, S., Barendsen, E., Schulte, C. (Eds.), *Computer Science Education: Perspectives on teaching and learning in school* (pp. 19–37). London: Bloomsbury Academic.

Heintz, F., Mannila, L., Färnqvist, T. (2016). A review of models for introducing computational thinking, computer science and computing in K–12 education. In: *Proceedings of the 2016 IEEE Frontiers in Education Conference (FIE)* (pp. 1–9). IEEE.

Hsu, T.-C., Chang, S.-C., Hung, Y.-T. (2018). How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers and Education*, 126, 296–310.

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., Werner, L. (2011). Computational Thinking for Youth in Practice. *ACM Inroads*, 2(1), 32–37.

Lonka, K. (2018). *Phenomenal learning from Finland*. Helsinki: Edita.

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., Peppler, K., Kafai, Y.B., Resnick, M., Rusk, N. (2008). Programming by choice: Urban youth learning programming with Scratch. In: Dougherty, J.D., Rodger, S. (Eds.), *Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08)* (pp. 367–371). New York, NY: ACM.

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

Mannila, L., Dagiene, V., Demo, B., Grgurina, N., Mirolo, C., Rolandsson, L., Settle, A. (2014). Computational thinking in K–9 education. In: Clear, A., Lister, R. (Eds.), *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference (ITiCSE '14)* (pp. 1–29). New York, NY: ACM.

Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2013). Learning computer science concepts with Scratch. *Computer Science Education*, 23(3), 239–264.

Moreno-León, J., Robles, G., Román-González, M. (2015). Dr. Scratch: Automatic analysis of Scratch projects to assess and foster computational thinking. *Revista de Educación a Distancia*, 15(46), 1–23.

Moreno-León, J., Robles, G., Román-González, M. (2017). Towards data-driven learning paths to develop computational thinking with Scratch. *IEEE Transactions on Emerging Topics in Computing.*

Román-González, M., Moreno-León, J., Robles, G. (2019). Combining Assessment Tools for a Comprehensive Evaluation of Computational Thinking Interventions. In: Kong, S.-C., Abelson, H. (Eds.), *Computational Thinking Education* (pp.79–98). Springer: Singapore.

Sáez-López, J.-M., Román-González, M., & Vázquez-Cano, E. (2016). Visual programming languages integrated across the curriculum in elementary school: A two year case study using "Scratch" in five schools. *Computers & Education,* 97, 129–141.

Seiter, L., Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. In: Simon, B., Clear, A., Cutts, Q. (Eds.), *Proceedings of the 9th annual international ACM conference on International computing education research (ICER '13)* (pp. 59–66). New York, NY: ACM.

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

Vihavainen, A., Vikberg, T., Luukkainen, M., Pärtel, M. (2013). Scaffolding students' learning using test my code. In: J. Carter (Eds.), *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)* (pp. 117–122). New York, NY: ACM.

Wangenheim, C., Hauck, J.C.R., Demetrio, M.F., Pelle, R., Cruz Alves, N., Barbosa, H., Azevedo, L.F. (2018). CodeMaster - Automatic assessment and grading of App Inventor and Snap! programs. *Informatics in Education*, 17(1), 117–150.

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

Wing, J. M. (2011). *A Definition of Computational Thinking from Jeannette Wing.* `https://computinged.wordpress.com/2011/03/22/a-definition-of-computational-thinking-from-jeanette-wing/`

Wilson, A., Hainey, T., Connolly, T.M. (2012). *Evaluation of computer games developed by primary school children to gauge understanding of programming concepts.* Paper presented at the 6th European Conference on Games-based Learning, Cork, Ireland.

Yin, R.K. (2012). *Case Study Research Design and Methods* (5th ed.). Thousand Oaks, CA: Sage.

J. Fagerlund (M.Ed.) is a doctoral student at the Department of Teacher Education, University of Jyväskylä, Finland, whose doctoral dissertation focuses on computational thinking through Scratch programming in the primary school context. He also operates as the regional coordinator in the Innokas Network (`http://innokas.fi/en`) in which he develops and trains teachers in ways to teach and learn 21st century skills with technology. ORCiD: `https://orcid.org/0000-0002-0717-5562` LinkedIn: `https://www.linkedin.com/in/jannefagerlund/` Postal address: Ruusupuisto, P.O. Box 35, 40014 University of Jyväskylä, Finland. E-mail: janne.fagerlund@jyu.fi Tel. +358408054711

P. Häkkinen is a Professor of educational technology at the Finnish Institute for Educational Research, University of Jyväskylä. Her research focuses on technology-enhanced learning, computer-supported collaborative learning and the progression of twenty-first-century skills (i.e., skills for problem solving and collaboration). ORCID: `https://orcid.org/0000-0001-6616-9114` LinkedIn: `https://linkedin.com/in/päivi-häkkinen-59132612` Postal address: Ruusupuisto, P.O. Box 35, 40014 University of Jyväskylä, Finland. E-mail: paivi.hakkinen@jyu.fi Tel. +358405843325

M. Vesisenaho is an adjunct professor, and a senior lecturer at the Department of Teacher Education, University of Jyväskylä. His background is in education, contextual design and computer science education. He has 20 years' experience in multidisciplinary education and research with national and international collaborators. His ambition is to innovatively reform learning with technology. ORCID: http://orcid.org/0000-0003-1160-139X Postal address: Ruusupuisto, P.O. Box 35, 40014 University of Jyväskylä, Finland. E-mail: mikko.vesisenaho@jyu.fi Tel. +358400247686

J. Viiri is a retired Professor of science and mathematics education at the Department of Teacher Education, University of Jyväskylä. He has taught physics in different educational levels. His research focuses on physics education, in particular, the use of models and representations in physics education, argumentation and communication between teachers and students. ORCID: http://orcid.org/0000-0003-3353-6859 Postal address: Ruusupuisto, P.O. Box 35, 40014 University of Jyväskylä, Finland. E-mail: jouni.p.t.viiri@jyu.fi Tel. +358505353611