Received: 19 December 2019 | Accepted: 19 April 2020

DOI: 10.1002/cae.22255

SPECIAL ISSUE ARTICLE

WILEY

Computational thinking in programming with Scratch in primary schools: A systematic review

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

Correspondence
Janne Fagerlund, Department of Teacher Education, University of Jyväskylä, Ruusupuisto, P.O. Box 35, 40014 Jyväskylä, Finland.
Email: janne.fagerlund@jyu.fi

Funding information
Emil Aaltosen Säätiö, Grant/Award Number: 170028 N1;
Keski-Suomen Rahasto, Grant/Award Number: 30161702

Abstract

Computer programming is being introduced in educational curricula, even at the primary school level. One goal of this implementation is to teach computational thinking (CT), which is potentially applicable in various computational problem-solving situations. However, the educational objective of CT in primary schools is somewhat unclear: curricula in various countries define learning objectives for topics, such as computer science, computing, programming or digital literacy but not for CT specifically. Additionally, there has been confusion in concretely and comprehensively defining and operationalising what to teach, learn and assess about CT in primary education even with popular programming akin to Scratch. In response to the growing demands of CT, by conducting a literature review on studies utilising Scratch in K–9, this study investigates what kind of CT has been assessed in Scratch at the primary education level. As a theoretical background for the review, we define a tangible educational objective for introducing CT comprehensively in primary education and concretise the fundamental skills and areas of understanding involved in CT as its "core educational principles". The results of the review summarise Scratch programming contents that students can manipulate and activities in which they can engage that foster CT. Moreover, methods for formatively assessing CT via students' Scratch projects and programming processes are explored. The results underpin that the summarised "CT-fostering" programming contents and activities in Scratch are vast and multidimensional. The next steps for this study are to refine pedagogically meaningful ways to assess CT in students' Scratch projects and programming processes.

KEYWORDS

assessment, computational thinking, primary school, programming, Scratch

1 | INTRODUCTION

The ubiquity of computing and computer science (CS) has expanded rapidly in modern society [1]. Meanwhile, countries such as Finland, England and Estonia have incorporated computer programming as a compulsory topic in primary education (K–9) [27,39]. Programming with Scratch, a graphical, block-based programming

language, is especially popular in this age group, thus providing a potentially impactful context for educational research. However, several scholars regard programming education not as an end in itself but essential—though nonexclusive—for fostering *computational thinking* (CT) (i.e., supporting the cognitive tasks involved in it) [23]. CT is an umbrella term that embodies an intellectual foundation necessary to understand the computational world and employ multidimensional problem-solving skills within and across disciplines [56,61].

Despite its popularity, there has been some shortcomings and uncertainty surrounding CT in terms of, for instance, teacher training needs concerning the aims and intents of CT education. In fact, curricula in different countries pose various educational objectives for such topics as CS, computing, programming or digital literacy but not for CT specifically [27]. Relatedly, there have been shortcomings in concretising what to teach, learn and assess regarding CT in schools, although previous literature portrays particular concepts and practices (e.g., "Algorithms", "Problem decomposition") that can shape students' skills and understanding in CT and contribute to its educational objective [8,34]. However, CT potentially learnt while programming with tools as Scratch has been typically perceived as, for instance, the code constructs that students use in their projects, which can be asserted to represent mere programming competence instead of the predictably higher level CT. When using such tools as Scratch, various programming contents that students manipulate and programming activities in which they engage can foster the skills and areas of understanding involved with CT in different ways. Previous literature has not systematically and thoroughly investigated how the practical programmatic affordances in Scratch can represent and foster the manifold skills and areas of understanding associated with CT as described in its core concepts and practices.

The aims of this study are to contextualise CT comprehensively in the Scratch programming environment for teaching and learning in primary school classrooms and explore the assessment of CT through Scratch in this context. In practice, a literature review for studies involving assessments in Scratch in K–9 is conducted. As a theoretical background, we define a tangible educational objective for CT in the context of programming in primary education based on previous literature. Moreover, as a springboard for investigating the skills and areas of understanding included in CT in Scratch, we concretise CT's *core educational principles* (CEPs)—fundamental computational facts, conceptual ideas, and techniques that students can learn—from CT concepts and practices presented in earlier research. The goals of the review are to gather Scratch programming contents and activities, use the CEPs as a lens to view them specifically as "CT-fostering" contents and activities, and explore ways in which they could be formatively assessed in classroom settings.

2 | COMPUTATIONAL THINKING THROUGH PROGRAMMING IN PRIMARY EDUCATION

2.1 | An educational objective

Wing [61,62] originally defined CT as "the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can effectively be carried out by an information-processing agent". Michaelson [43] underlined that CT is a way of understanding problems whereas CS provides concepts for CT in search of a praxis. Aho [1] revisited Wing's original definition and emphasised that solutions pertinent to CT are namely algorithmic. However, CT still has no solid core definition [24]. It has been viewed as a competence [58], a thought process [1,62], a set of skills [61] and a problem-solving process [54]. However, the consensus is that it draws on disciplinary concepts and models central to CS and utilises the power of computing [56].

The purpose of primary education is to learn about the world and to prepare for subsequent studies and working life. Although CT's transferability across problem-solving contexts has been questioned [14], Wing [61] posited that CT as a collection of transversal skills and knowledge is necessary for everyone. Lonka et al [33] underlined that students, regardless of their future profession, should learn to identify the central principles and practices of programming and understand how they influence everyday life.

To include CT's such essential characteristics and purposes [33,53,56,61] tangibly in primary education, we define the following educational objective for it: students learn to understand what computing can/cannot do, understand how computers do the things that they do and apply computational tools, models and ideas to solve problems in various contexts. According to recent reviews of curricula in various countries, such educational ideas are relevant in schools via CS education, programming or embedded within different subjects, but not for CT specifically [27,39]. By exploring computing, students should also gain certain attitudes and perspectives, such as understanding computational ethics [33]. However, this study limits its scope by focusing on CT's key concepts and practices, which have been often highlighted in previous literature to characterise

fundamental areas of understanding in computing and skills in computational problem-solving.

Definitions for the key concepts and practices in CT have varied throughout previous literature. For instance, in the context of Scratch, Brennan and Resnick [9] presented a concrete CT framework that comprised concepts (e.g., loop, variable), practices (e.g., debugging, iteration) and perspectives (e.g., expressing, questioning). Although meaningful for CT, such context-specific frameworks may be unsuitable for framing CT across programming contexts and promoting deeper learning. [24] Therefore, based on prior research framing CT concepts and practices in a broader fashion, we concretise the fundamental skills and areas of understanding involved in CT as its *core educational principles* (CEPs) as a background.

2.2 | Core educational principles

Several studies have framed CT's key concepts and practices more generally in programming, computing or CS in various ways. CT is an elusive term that continues finding clear borders, and it involves areas that could be interpreted to be more in its "central" or "peripheral zones". Concise views of CT can be rather programming-centric and omit potentially essential areas in the general-level CT. In turn, generous views may overlap with other competence areas, such as math. By framing our view of CT based on several previous works, we strive to adopt a relatively generous rather than a concise view. The motivation is that the more generous views have been adopted less often, and they can expand our understanding of the potentially meaningful borders of CT assessment through Scratch in K–9 and be feasibly reduced to the extent, as needed.

Settle and Perkovic [51] developed a conceptual framework to implement CT across the curriculum in undergraduate education. In 2009, the International Society for Technology in Education and the Computer Science Teachers Association [3] devised an operational definition for CT concepts and capabilities to promote their incorporation in K–12 classrooms. In the aftermath of computing having been introduced in British schools in 2014, Czismadia et al [13] developed a framework for guiding teachers in teaching CT-related concepts, approaches and techniques in computing classrooms. Relatedly, Angeli et al [2] designed a K–6 CT curriculum comprising CT skills and implications for teacher knowledge. To demystify CT's ill-structured nature, Shute et al [53] reviewed CT literature and showed examples of its definitions, interventions and assessments in K–12. Similarly, Hsu et al [28] reviewed prior literature and discussed how CT could be taught and learned in K–12. To further illuminate CT's application in different contexts, Grover and Pea [24] elaborated what concepts and practices CT encompasses.

To concretise the skills and areas of understanding associated with CT concepts and practices in these works as atomic elements to enable their systematic contextualisation in Scratch, the definitions of the concepts and practices can be summarised to include CT's CEPs for teaching and learning at the primary school level.

- *Abstraction.* A range of digital devices can be computers that run programmes [13,24]. Programming languages, algorithms and data are abstractions of real-world phenomena [13,24,28]. Solving complex problems becomes easier by reducing unnecessary detail and by focusing on parts that matter (via, e.g., using data structures and an appropriate notation) [2,13,24,28].
- *Algorithms.* Programmers solve problems with sets of instructions starting from an initial state, going through a sequence of intermediate states and reaching a final goal state [2,3,13,24,28,51,53]. Sequencing, selection and repetition are the basic building blocks of algorithms [2,3,13,24]. Recursive solutions solve simpler versions of the same problem [3,13,24].
- *Automation.* Automated computation can solve problems [13,24,28]. Programmers design programmes with computer code for computers to execute [13,24,51]. Computers can use a range of input and output devices [13].
- *Collaboration.* Programmers divide tasks and alternate in roles [24]. Programmers build on one another's projects [2,24]. Programmers distribute solutions to others [24].
- *Coordination and Parallelism.* Computers can execute divided sets of instructions in parallel [3,13,28,53]. The timing of computation at participating processes requires control [51].
- *Creativity.* Programmers employ alternate approaches to solving problems and "out-of-the-box thinking" [24]. Creating projects is a form of creative expression [24].
- *Data.* Programmers find and collect data from various sources and multilayered datasets that are related to each other [3,28,53]. Programmes work with various data types (e.g., text, numbers) [3,13,28]. Programmes store, move and perform calculations on data [2,3,13,51]. Programmes store data in various data structures (e.g., variable, table, list, graph) [2,3,13].
- *Efficiency.* Algorithms have no redundant or unnecessary steps [13,53]. Designed solutions are easy for people to use [13]. Designed solutions work effectively and promote positive user experience [13,24].

Designed solutions function correctly under all circumstances [13,24].

- *Iteration.* Programmers refine solutions through design, testing and debugging until the ideal result is achieved [24,53].
- *Logic.* Programmers analyse situations and check facts to make and verify predictions, make decisions and reach conclusions [2,13,24]. Formulated instructions comprise conditional logic, Boolean logic, arithmetic operations and other logical frameworks [2,13,24,28].
- *Modelling and design.* Programmers design human-readable representations and models of an algorithmic design, which could later be programmed [13,24,28,53]. Programmers organise the structure, appearance and functionality of a system well [13,51]. Visual models, simulations and animations represent how a system operates [2,3,13,28].
- *Patterns and Generalisation.* Data and information structures comprise repeating patterns based on similarities and differences in them [2,13,24,28,53]. Repeating patterns form general-level solutions that apply to a class of similar problems [3,13,24,28,53]. General-level ideas and solutions solve problems in new situations and domains [13,24,28,53].
- *Problem decomposition.* Large problems and artefacts decompose into smaller and simpler parts that can be solved separately [2,13,24,28,53]. Large systems are composed of smaller meaningful parts [2,24]. Programmes comprise objects, the main programme and functions [3].
- *Testing and debugging.* Programmers evaluate and verify solutions for appropriateness according to their desired result, goal or set criteria [2,13,24,28]. Programmers evaluate solutions for functional accuracy and detect flaws using methods involving observation of artefacts in use and comparing similar artefacts [2,13,24,28,53]. Programmers trace code, design and run test plans and test cases and apply heuristics to isolate errors and fix them [2,13,24,28,53]. Programmers make fair and honest judgements in complex situations that are not free of values and constraints [13].

In practice, various programming tasks can foster skills and understanding in the ways of thinking and doing involved in CT as described in the CEPs. In Scratch, students manipulate programmatic contents, that is, the objects and logic structures that establish computational processes in their projects, and engage in certain programming activities while designing said contents [9]. Hence, it is meaningful to examine how various Scratch programming contents and activities contextualise the CEPs in practice.

2.3 | Assessment in scratch

Scratch is a free web-based programming tool that allows the creation of media projects, such as games, interactive stories and animations, connected to young peoples' personal interests and experiences. Projects are designed by combining graphical blocks to produce behaviours for digital characters ("sprites"). Block-based languages typically have a "low floor": students cannot make syntactic mistakes because only co-applicable blocks combine into algorithmic sets of instructions ("scripts") [9,38].

Despite the affordances of graphical tools, programming is cognitively complex, and rich conceptual mental models may not emerge spontaneously [4,40]. An "in time" pedagogy in which new knowledge is presented whenever necessary through various project-based activities is a popular approach; however, it requires the careful formulation of authentic problems and selection of projects (i.e., ways to introduce CT appropriately via programming contents and activities) [20,34]. Moreover, learning can be supported with a formative assessment that determines "where the learner is going", "where the learner is right now" and "how to get there". In practice, instructors should clarify the intentions and criteria for success, elicit evidence of students' understanding and provide appropriate feedback that moves learning forward [6]. Programming is a potentially fruitful platform for enabling these processes because it demonstrates students' CT and provides a potential accommodation for timely and targeted learning support [23,34].

Several previous empirical studies have shown in part how specific programming contents and activities in Scratch could be assessed. However, the contents and activities have been scarcely contextualised in CT. To examine how CT could be thoroughly introduced and respectively assessed in Scratch in K–9 (primary education), this study reviews prior literature focused on assessing Scratch contents and activities in K–9 and aligns them to CT concepts and practices according to the summarised CEPs (see Section 2.2). The purpose is to derive elementary CT-fostering learning contents and activities and to explore appropriate methods for their formative assessment in primary schools. Hence, the research questions are:

What Scratch programming contents and activities have been assessed in K–9?

How have Scratch programming contents and activities been assessed?

How do different Scratch programming contents and activities contextualise CT concepts and practices via the CEPs?

3 | METHODS

3.1 | Search procedures

To begin answering the research questions, literature searches were performed for peer-reviewed studies focusing on the assessment of Scratch programming contents and activities in K–9 (Figure 1). First, searches were conducted with the terms "computational thinking" and "Scratch" in the ScienceDirect, ERIC, SCOPUS and ACM databases. Publications were sought as far back as 2007 when Scratch was released [9]. The searches resulted in 432 studies (98 in ScienceDirect, 27 in ERIC, 217 in SCOPUS and 90 in ACM) on November 27th, 2019. Duplicate and inaccessible publications were excluded from this collection.

The abstracts of the remaining studies were screened, and both empirical and nonempirical studies were included if they addressed assessment in Scratch (or highly similar programming languages) in K–9. Publications conceptualising generic assessment frameworks were included if Scratch and primary education were mentioned as potential application domains. Studies set in other or unclear educational levels were excluded to maintain a focus on primary schools. Studies written in other languages than English were excluded.

The remaining 50 studies was not presumed to cover all potentially relevant work. Further searches were conducted similarly with the terms "computational thinking" and "Scratch" on Google Scholar, which provided a running list of publications in decreasing order of relevance. These publications were accessed individually until the search results concluded to no longer provide relevant studies. Simultaneously, the reference lists of all included studies were examined for discovering other potentially relevant publications.

Altogether 81 obtained studies were then screened for the assessment instruments that they employed. Studies analysing students' Scratch project contents or their programming activities in Scratch were included. Studies analysing the learning of other subject domain contents or addressing other theoretical areas such as motivation, attitudes and misconceptions were excluded. Assessment instruments that were defined in insufficient detail or were adapted in an unaltered form from prior studies were excluded since they provided no additional information for the RQs. For example, we found that several articles employed the assessment instrument called "Dr. Scratch" (see results). To attain information regarding what Scratch programming contents and activities have been assessed in K–9 and how said contents and activities have been assessed altogether, we only included the paper that originally introduced said contents and activities, granted that the work was attainable. Finally, 30 publications were selected for review.

3.2 | Analysis of studies

The Scratch programming contents and activities assessed in the studies were described based on their type (RQ1)

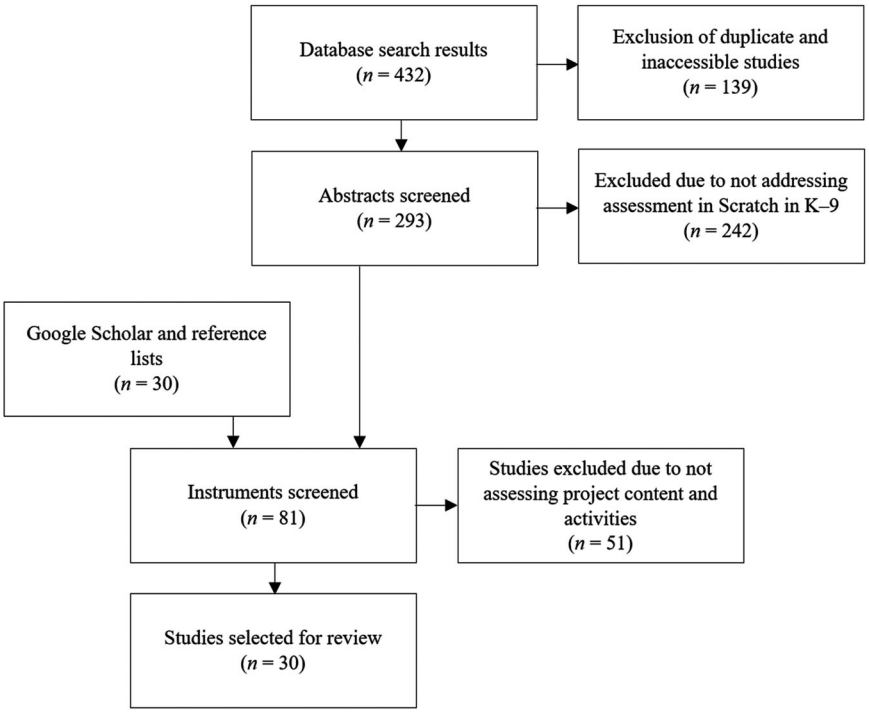

FIGURE 1 Literature search protocol

and the employed assessment method and taxonomy or rubric (RQ2). Simultaneously, by employing content analysis, the contents and activities were aligned to CT concepts and practices according to the CT's CEPs (see Section 2.2) that they contextualised (RQ3) (indicated in results by CT concepts and practices highlighted in parentheses). The analysis was carried out by the first author.

Due to the complexity of CT, however, there is an immense level of detail to which the contextualisation in RQ3 could potentially reach. For instance, reducing unnecessary detail (Abstraction) can involve various broader programming tasks and detailed subtasks. However, Voogt et al [58] stated that it is important to discover "what matters" for CT. Therefore, as our first step, we settled on merely describing what the assessed contents and activities that contextualised CT were instead of attempting to further analyse how they could foster CT in different ways.

The analysis resulted in rubrics to Scratch contents and activities that foster skills and understanding in CT concepts and practices. The discovered assessment methods were examined according to how they potentially enabled formative assessment processes as presented by Black and Wiliam [6].

Potential limitations in reviews especially concern the definition of the RQs, search procedure, selection of articles, bias in the source material and its quality and the ways of presenting the results [26]. Therefore, we wish to make the following remarks concerning the repeatability, objectivity and transparency herein. By describing the procedure comprehensively and in detail, we aimed to reveal any bias (e.g., concerning the use of appropriate search strings in representative databases) [12,26]. Additionally, we strived to describe the inferences made and the logic behind them clearly and give equal weight to all reviewed work, though spotlighting evidence that stands out in the process and potentially suggests subjectivity in the source material [26]. Furthermore, we aimed to reinforce consistency in the analysis by iteratively evaluating the contents of the articles, ensuring that we interpreted them the same way at different times [35]. By externally checking the research process and debriefing the results among the authors, we aimed to verify further that the meanings and interpretations resonated among different researchers [12].

4 | FINDINGS

4.1 | Scratch contents and activities and their assessment

Prior studies utilising Scratch in K–9 involved the assessment of various programming contents and activities with diverse assessment methods and taxonomies or rubrics (RQ1, RQ2) (Table 1). Four distinct programming substance categories were found and were named as "code constructs", "coding patterns", "programming activities" and "other programming contents". Altogether, 20 studies assessed *code constructs* as the logic structures (e.g., sequence of blocks, "repeat" [44]) that programmers use to establish algorithmic sets of instructions in Scratch projects. Ten studies assessed *coding patterns*, combinations of code constructs that act as larger programmatic units for specific semantical purposes (e.g., "Animate Motion" [50]). Eleven studies examined students' *programming activities* (e.g., "script analysis" [30]), whereas six studies examined *other programming contents* (e.g., "project genres" [19]). Only six studies considered the direct assessment of CT, and the remaining studies assessed the contents or activities with or without presenting CT as a motivational theme.

Structured with the aforementioned four substance categories, the following subsections describe the nature of the discovered contents and activities and their assessment methods more completely and elaborate their relationships with the CEPs (RQ3).

4.2 | CT's CEPs in Scratch

4.2.1 | Code constructs

Three studies assessing code constructs examined CT specifically. "Dr. Scratch", a web-based automatic analysis tool, assessed the use of blocks in Scratch projects (Table 2) [44]. Relatedly, Wangenheim et al [59] used "CodeMaster", a similar yet more extensive rubric for projects made in the Snap! programming environment. In terms of CEPs contextualised in Scratch by these tools, for instance, "if" blocks and logic operations contextualise conditional logic and Boolean logic (Logic), and the rubrics to "flow control" contextualise the basic building blocks of algorithms (Algorithms). Moreover, the rubrics to "data representation" contextualise working with different data types, performing operations on data and using various data structures (Data) in addition to abstracting real-world phenomena as data (Abstraction). Moreover, the "ANTLR" tool presented by Chang et al [11] expanded the rubrics of Dr. Scratch to include recursion (Algorithms).

Two other automated tools, "Ninja Code Village" (NCV) presented by Ota et al [46] and "Scrape" by used by Ke [30], examine similar code constructs to Dr. Scratch without aligning them to CT. However, similar to Dr. Scratch's rubrics in "Abstraction and Problem decomposition", NCV's rubrics for the "procedure"

TABLE 1 A summary of studies involving the assessment of Scratch programming contents and activities in K–9

		Assessment in Scratch		
#	Authors	Contents/activities	Method	Taxonomy/rubric
1	Benton et al [5]	Coding patterns (CT)	Self-evaluation	Difficulty rating
2	Blau et al [7]	Other programming contents	Artefact analysis	Presence/frequency
3	Brennan and Resnick [9]	Code constructs + programming activities (CT)	Artefact analysis Performance evaluation Interview	Presence/frequency Skill description
4	Burke [10]	Code constructs Programming activities	Artefact analysis Observation Interview	Presence/frequency Description, data-driven
5	Chang et al [11]	Code constructs (CT)	Artefact analysis	Presence/frequency
6	Ericson and McKlin [15]	Code constructs Coding patterns	Test	Correct answer Correct drawing
7	Franklin et al [16]	Coding patterns Code constructs Programming activities	Observation Test Observation	Correctness level Correct answer Behaviour type
8	Franklin et al [17]	Code constructs Coding patterns	Artefact analysis	Content completion (percentage)
9	Funke et al [19]	Coding patterns Code constructs Other programming contents	Artefact analysis	Progression level Presence/frequency
10	Funke and Geldreich [18]	Code constructs	Log data analysis	Description
11	Grover and Basu [21]	Code constructs Coding patterns	Test Think-aloud	Correct response
12	Gutierrez et al [25]	Other programming contents	Artefact analysis	Presence/frequency
13	Israel et al [29]	Programming activities	Observation + discourse analysis	Behaviour type
14	Ke [30]	Code constructs Programming activities	Artefact analysis Observation	Presence/frequency Behaviour type
15	Lewis [31]	Code constructs	Test Self-evaluation	Correct answer Likert
16	Lewis and Shah [32]	Programming activities	Discourse analysis	Behaviour type Hypotheses, data-driven
17	Mako Hill et al [36]	Programming activities Other programming contents	Artefact analysis	Presence/frequency
18	Maloney et al [37]	Code constructs	Artefact analysis	Presence/frequency
19	Meerbaum-Salant et al [41]	Programming activities	Observation	Behaviour type
20	Meerbaum-Salant et al [42]	Code constructs Coding patterns	Test	Correct response
21	Moreno-León et al [44]	Code constructs (CT)	Artefact analysis	Presence
22	Ota et al [46]	Coding patterns Code constructs	Artefact analysis	Presence
23	Sáez-López et al [55]	Code constructs Programming activities + other programming contents	Test Self-evaluation Observation	N/A Performance level

(Continues)

TABLE 1 (Continued)

#	Authors	Assessment in Scratch		
		Contents/activities	**Method**	**Taxonomy/rubric**
24	Seiter [49]	Coding patterns	Artefact analysis	Presence
25	Seiter and Foreman [50]	Code constructs + coding patterns (CT)	Artefact analysis	Presence
26	Shah et al [52]	Programming activities	Discourse analysis	Behaviour type
27	Tsan et al [57]	Programming activities	Discourse analysis Observation	Behaviour type
28	Wangenheim et al [59]	Code constructs (CT)	Artefact analysis	Presence
29	Wilson et al [60]	Code constructs Other programming contents	Artefact analysis	Presence
30	Zur-Bargury et al [63]	Code constructs	Test	Correct response

code construct contextualise different kinds of functions and procedures that act as separate instruction sets to solve specific problems (Algorithms). Moreover, Scrape and Dr. Scratch examined external device usage via various input/output devices (e.g., keyboard, mouse) (Automation).

Regarding other assessment methods, Lewis [31] asked students to describe the output of example scripts comprising certain code constructs and evaluate how hard it was to learn them. Meerbaum-Salant et al [42] conducted summative tests with a revised Bloom/SOLO taxonomy on students' understanding in parallel execution within and across different sprites, which was underlined to often require the synchronisation of different scripts. Relatedly, several other studies [10,19,37,60] manually examined students' projects for the "synchronisation" code construct, which was juxtaposed with the "coordination" or "communication" code constructs. The implementation of synchronisation, coordination and communication contextualises controlling the timing of computation in participating processes (Coordination and Parallelism). In Scratch, coordination and synchronisation of parallel processes can occur with timing (e.g., the "wait" block), state-sync (e.g., the "wait

TABLE 2 Evidence for CT as examined by Dr. Scratch [26]

CT concept	Competence level		
	Basic	**Developing**	**Proficient**
Abstraction and Problem decomposition	More than one script and more than one sprite	Make-a-blocks	Cloning
Parallelism	Two scripts start on "green flag"	Two scripts start on when key is pressed/when sprite is clicked on the same sprite	Two scripts start on "when I receive message", "create clone", "when %s is >%s" or "when backdrop change to" blocks
Logical thinking	"If" block	"If-else" block	Logic operations
Synchronisation	"Wait" block	"Broadcast", "when I receive message", "stop all", "stop program" or "stop programs sprite" blocks	"Wait until", "when backdrop change to" or "broadcast and wait" blocks
Flow control	Sequence of blocks	"Repeat" or "forever" blocks	"Repeat until" block
User interactivity	"Green flag" block	"Key pressed", "sprite clicked", "ask and wait" or mouse blocks	"When %s is >%s", video or audio blocks
Data representation	Modifiers of sprite properties	Operations on variables	Operations on lists

until" block) or event-sync (e.g., the "when I receive" block) and by blocking or stopping further script execution [44,50]. Moreover, Franklin et al [16,17] manually assessed the use of the "initialisation" code construct, that is, setting initial state values (Algorithms) for sprite properties such as location or size.

4.2.2 | Coding patterns

Seiter and Foreman [50] developed the "Progression for Early Computational Thinking" (PECT) model to manually examine CT through project-wide design pattern variables: "Animate Looks", "Animate Motion", "Conversate", "Collide", "Maintain Score" and "User Interaction". The design pattern variables are assessed with rubrics to specific code construct combinations, whereas students' understanding in CT is indicated by the presence of specific level variables in a Scratch project. In addition to the relationships between CT and programming contents disclosed directly in PECT (see Seiter and Foreman [50] for detailed rubrics), in Scratch, coding patterns and code constructs themselves contextualise repeating patterns and generalisable computational solutions (Patterns and Generalisation). The implementation of coding patterns and code constructs also contextualises breaking complex projects into smaller, manageable parts that establish the larger system. Coding patterns could also be considered as the functions of different objects (i.e., sprites) (both Problem decomposition). Moreover, each coding pattern can be interpreted as a separate solution to a problem (Algorithms), which, in turn, is an abstraction of a real-world phenomenon (Abstraction).

Benton et al [5] asked students to rate the difficulty of different kinds of algorithms, which resembled PECT's "Animate Motion" coding pattern. Franklin et al [17] examined the "Breaking down actions" coding pattern, which resembled a combination of PECT's "Collision" and "Animate Motion". However, unlike in PECT, this coding pattern required parametric precision (e.g., an exact number in a "move" block), which can be essential in ensuring that designed solutions achieve the desired results (Efficiency). Similarly, test questions employed by Meerbaum-Salant et al [42] and Grover and Basu [21] concerning coding patterns, which resembled PECT's "Animate Motion" and "Maintain Score", necessitated distinguishing between separate overlapping coding patterns (see example in Figure 2). These solutions spotlighted the option of examining individually instantiated rather than project-wide coding patterns in students' projects.

Ericson and McKlin [15] asked students to draw the outputs of scripts comprising a coding pattern, which resembled PECT's "Animate Motion" with the "pen" code construct. In Scratch, pen is used to draw visual lines as sprites move and, therefore, visualise algorithms (Modelling and design), although several other programmed features (e.g., conversations, animations) also manifest visually or vocally in Scratch. The authors also introduced a coding pattern for reading keyboard inputs and storing them in the "answer" variable (Automation) in addition to using conditional structures and Boolean expressions to evaluate the value stored in the variable (Logic).

Franklin et al [16] adopted a mixed methods approach with the "Hairball" plugin and a qualitative coding scheme to additionally examine the "Complex Animation" coding pattern, which resembled PECT's "Animate Motion" and "Animate Looks" with a "loop" code construct. Similarly, Seiter [49] used a three-level SOLO taxonomy to assess a "Synchronising costume with motion" coding pattern, which resembled the parallel execution of the same two coding patterns. Additionally,

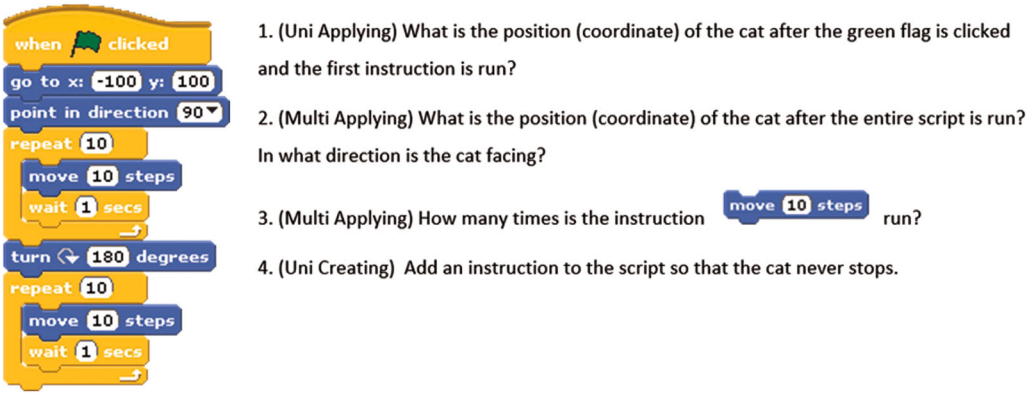

FIGURE 2 Questions that necessitate distinguishing two independent motion parameters: facing direction and location (supplementary materials by Meerbaum-Salant et al 2013)

the "Multi-sprite conversation" coding pattern encompassed a synchronised dialogue-animation. The synchronisation of coding patterns themselves also contextualises controlling the timing of participating processes (Coordination and Parallelism).

4.2.3 | Other programming contents

Blau et al [7] and Mako Hill et al [36] examined the amount of scripts and sprites in students' projects. Similarly with Dr. Scratch, Moreno-León et al [44] examined "more than one script and one sprite" aligned to Abstraction, which encompasses solving complex problems, and Problem decomposition, which encompasses decomposing a complex system into manageable parts. Relatedly, Gutierrez et al [25] examined "documentation" (i.e., code comments) in projects whereas Wilson et al [60] and Funke et al [19] examined the "custom naming of sprites", "meaningful naming of variables" and "no extraneous blocks", all of which can make complex artefacts more understandable and manageable (Abstraction) and organise their structure and appearance (Modelling and design). Additionally, these studies examined the "functionality of projects", which contributes to ensuring that a project is correct with respect to the desired goals (Efficiency). A "clearly defined goal" and "instructions" as also examined by these studies are key features in projects that are easy to use and trigger appropriate user experiences (Efficiency). Then again, "customised sprites", "customised stages", "originality of a project" and the "ability to communicate and express through artefacts", as examined by Sáez-López et al [55], can promote creative expression (Creativity).

Lastly, Blau et al [7] examined how many projects students had created and remixed while Funke et al [19] categorised projects' genres. Gutierrez et al [25] examined the extent to which students had made only superficial changes with respect to sample projects. Designing and remixing a number of projects contributes to creating different kinds of computerised solutions that each have a specific purpose (Automation).

4.2.4 | Programming activities

None of the 11 studies that examined programming activities focused directly on CT apart from Brennan and Resnick [9], who described four practices – "being incremental and iterative", "testing and debugging", "reusing and remixing" and "abstracting and modularising" – which largely aligned with the broader CT concepts and practices as examined in the current work. They also proposed two methods for examining said practices: interviews and design scenarios. Similar to Brennan and Resnick's "reusing and remixing", Blau et al [7] examined students' social participation (e.g., friends, comments and favourited projects), whereas Mako Hill et al [36] examined students' credit-giving habits. These activities relate to building on other programmers' work and distributing one's own work (Collaboration).

Focusing on project design phases, Burke [10] categorised students' programming processes into "brainstorming and outlining" and "drafting, feedback and revising". Ke [30] categorised students' game development acts more elaborately (e.g., "Off-task", "Script analysis", "Test play"). Funke and Geldreich [18] conceptualised a visualisation technique to describe script design processes. Meerbaum-Salant et al [41] identified two programming habits: bottom-up programming (bricolage) and extremely fine-grained programming. These activities demonstrate different ways to plan (Modelling and design) and refine solutions (Iteration) and evaluate them, detect flaws, isolate errors and fix bugs (Testing and debugging).

Focusing on human-to-human interactions, Franklin et al [16] recorded the help levels students required when programming. Israel et al [29] developed the C-COI instrument for coding students' behaviours as steps in collaborative problem-solving processes. Shah et al [52] and Lewis and Shah [32] examined students' equity, quality of collaboration, task focus and speech during programming. Sáez-López et al [55] questioned and observed students' sharing and playing with their programmes, active participation and clear communication. Tsan et al [57] analysed students' collaborative dialogue. Such manifold aspects of interaction affect task division and role alternating (Collaboration).

5 | DISCUSSION

5.1 | Typifying elementary CT in Scratch

By conducting a literature review, we explored the assessments of programming contents and activities in Scratch and aligned them to CT concepts and practices according to CT's CEPs (in Section 2.2), which were derived from previous contemporary literature as a background to enable the systematic contextualisation of CT in Scratch. The view of CT adopted in this study is relatively broad, and it can encompass areas that can be positioned in a more "central" or "peripheral zones" of CT and get included or excluded as needed. In the following sections, we provide summaries that include the

reviewed CT-fostering Scratch programming contents and activities. As encouraged by prior studies [23,34], we also discuss the formative assessment of the contents and activities in students' authentic programming projects and processes rather than, for instance, ranking or certifying students' competence or regarding them with tests to highlight potentially meaningful ways to support learning.

The summaries should not be regarded as complete since CT is a developing body of broad and complex ideas. Hence, we also discuss which CEPs were not straightforwardly contextualised in Scratch. Additionally, as CT is a collection of holistic skills and understanding in computational problem-solving [56,61], the contents and activities could be interpreted to contextualise different areas in CT in various ways. Therefore, we recap and capsulise the results mainly as Scratch contents and activities contextualising the CT concepts and practices more generally rather than the single CEPs. Moreover, the contents and activities should not be viewed as isolated gimmicks but as components that conjoin meaningfully while, for instance, designing games, creating storytelling projects or animating while processing learning contents in other curricular areas [20,45]. Scratch can promote self-expression, interest and fun in learning programming in settings that are built on such pedagogical underpinnings as constructionism and co-creation [9,47]. Meaningful learning thereby includes authentic problems and meaningful selections of projects. In terms of CT in such settings, it is important to focus especially on how students are thinking as they are programming [34].

5.1.1 | Contents in Scratch projects

Students' CT can be evaluated based on the code constructs (e.g., "loop", "variable"), coding patterns (e.g., "change location") and other programming contents (e.g., sprite naming) (Table 3) they have implemented in different kinds of Scratch projects. The PECT model presented by Seiter and Foreman [50] proposed a comparatively comprehensive rubric for coding patterns and code constructs. However, parametric precision highlighted the importance of examining individually instantiated patterns rather than project-wide coding patterns: for instance, each property (e.g., size, position) of each sprite has an independent state, which necessitates paying attention to, for instance, initialising them separately (e.g., "change location for Sprite1") [16,17]. The presence, frequency, correct implementation or completion rate of particular contents as evaluated in several prior studies can demonstrate students' CT. Although particular studies [5,19,42] additionally proposed progression levels or difficulty ratings for particular contents, fully congruent and thus conclusive learning progressions for CT in Scratch were not explicit in the reviewed studies. Therefore, applying a learning taxonomy (e.g., Bloom/SOLO [42]) systematically to the contents gathered herein would require further investigation.

5.1.2 | Activities in Scratch

CT-fostering Scratch programming activities may leave traceable evidence in projects as static contents but may be more thoroughly identified in students' programming processes. For example, Standl [54] framed CT as a problem-solving process that includes phases, such as describing the problem, abstracting the problem, decomposing the problem, designing the algorithm and testing the solution. The CT-fostering activities in Scratch described in the reviewed studies can be similarly summarised as a model of a CT problem-solving process (Figure 3). As demonstrated by several studies, students' CT-fostering activities can be evaluated by means of observation, interviewing or self-evaluation next to a desired skill description or performance level.

In particular, project planning can include, for instance, algorithmic flowcharts, pseudo-code, drawings and lists (Modelling and design) [10]. Decomposition of planned or programmed solutions into smaller, manageable parts (Problem decomposition) could be examined with a rubric to coding patterns and code constructs, such as with the one presented by Seiter and Foreman [50]. The actual code-writing can resemble "bricolage" or decomposition into logically coherent units, and it can comprise repeating cycles of designing, analysing scripts and testing play (Iteration, Testing and debugging) [30,41]. However, due to lack of empirical demonstration, it is somewhat unclear what kinds of activities in Scratch lead to effective and fair evaluation and verification of programmed solutions (testing and debugging) and removing redundant and unnecessary steps in scripts (Efficiency). Meanwhile, solutions can be shared and remixed (Collaboration) to gain feedback and new ideas [9]. Additionally, during programming, students may recognise how previously designed coding patterns or code constructs could be reused (Patterns and Generalisation), although it remains somewhat unclear how such events occur in practice. Furthermore, task division and role alternating (Collaboration), which may be influenced by factors concerning equity, task focus, talk, active participation and clear communication, are present during all activities [32].

TABLE 3 CT-fostering programming contents in Scratch projects

CT concept/ practice	Scratch contents (and source studies, see Table 1)
Abstraction	• Sprite properties, variables and lists (abstractions of properties) [1,3,4,6,8,9,11,14,15,18,20,22,24,25,29,30] • Coding patterns, make-a-blocks and cloning (abstractions of behaviours) [1,5–9,11,14,20–22,24,25,28,30] • Continuous events (repeat until), discrete events (wait until) and initialisation (abstractions of states) [1,7,8,11,20,22,24,25,28,30] • Complex projects with several scripts and sprites [3,5,9,21,28]
Algorithms	• Coding patterns, make-a-blocks and cloning (coding separate procedures as specific functionalities) [1,6–9,11,14,20,22,24,25,28,30] • Initialisation [1,7,8,20,24,25] • Sequencing, looping and selection in coding patterns (algorithm control) [1,3–6,8,9,11,14,15,18,20–25,28–30] • Self-calling (recursive) make-a-blocks [5,22]
Automation	• Green flag, key press, sprite click, keyboard input, mouse, sensing, video and audio events (I/O device use) [3–7,9,14,18,21,22,24,25,28–30] • Animations, games, art, stories and simulations (project genres) [3,9]
Collaboration	• Publishing projects [2] • Remixing and credit-giving [3,17] • Commenting, requesting friends, favouriting, “love-its” [2,3]
Coordination and Parallelism	• Synchronised parallel code constructs and coding patterns within a sprite and across sprites [3–5,9,12,18,20–25,28,29] • Coordinated parallel code constructs and coding patterns with timing, states, events, blocking (ask and wait) and stopping script execution [3–9,18,20–22,24,25,28,29]
Creativity	• Customised sprites and stages [9,12,23] • Modifying a remixed project [3,12,29] • Expressing personal interest areas [3,23]
Data	• Sprite properties, Scratch variables, custom variables, lists and cloud variables (storing and manipulating data in data types) [1,3,4,6,8,9,11,14,15,18,20,22,24,25,28–30]
Efficiency	• Precise data manipulation [1,6,8,15,20,24,30] • Defined project goal [29] • Use instructions [9,29] • Functionality [9,29]
Logic	• If, if-else, nested conditionals [3,4,6,9,11,14,15,18,20–22,28–30] • And, or, not (Boolean logic) [3–5,9,11,14,15,18,21,28,29] • Arithmetic operations [3,9,15,18,29,30] • Absolute and relational operations [1,3]
Modelling and design	• Looks and motion animation, pen drawing and sounds (algorithm animation) [1,3–9,11,14,15,22,24,28,30] • No extraneous blocks [9,12,29] • Meaningful names for sprites and variables [9,11,12,29,30] • Code comments [12]
Patterns and Generalisation	• Reinstantiated code constructs [4,7,9,25] • Reinstantiated coding patterns [7,11,20,30]
Problem decomposition	• Coding patterns and code constructs (decomposition) [1,3–11,14–16,18,20–25,28–30] • Separately scripted behaviours or actions (modularisation) [3,25]

Note: The concepts and practices may not be entirely mutually exclusive in terms of the contents.

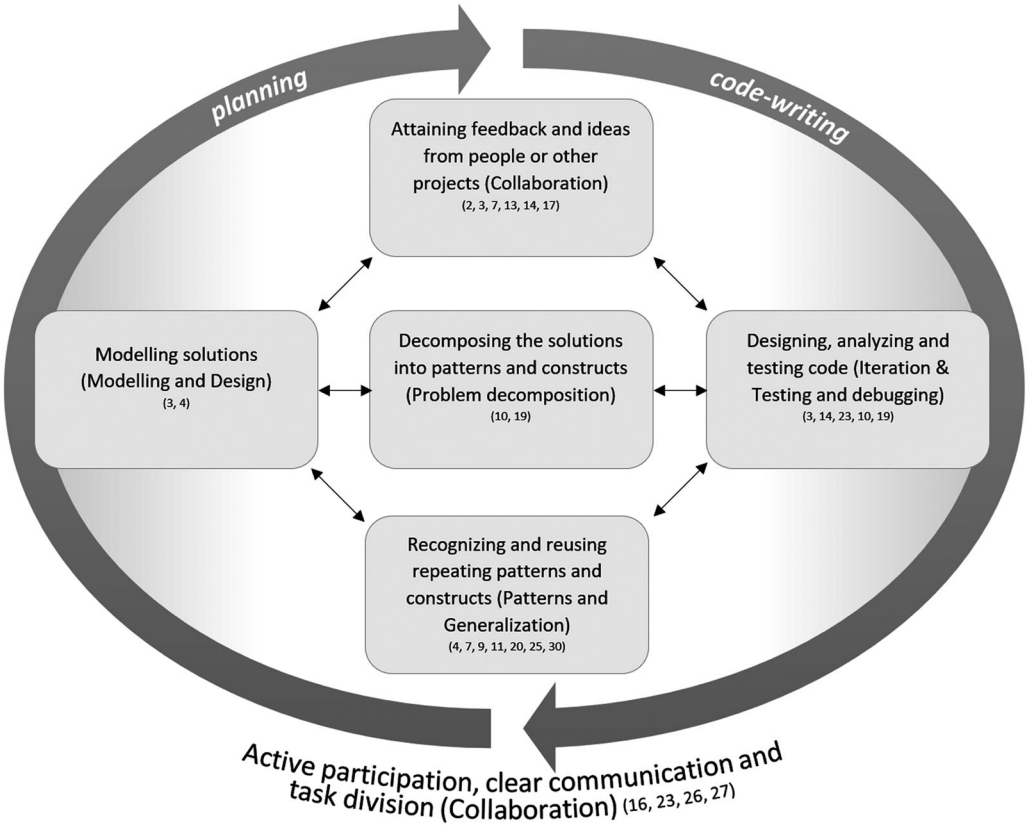

FIGURE 3 CT-fostering activities in Scratch (and source studies, see Table 1)

5.2 | Formative assessment of CT in Scratch

In this study, we lean on the following notion on formative assessment: its processes involve (1) clarifying learning intentions and criteria for success, (2) eliciting evidence of students' current understanding and (3) providing feedback to move learning forward [6].

In CT, holistic assessment should recognise the diversity of problem-solving situations and align contextualised, task-specific assessment rubrics to the focal areas of CT (1) [22,45]. Educators could utilise concrete and contextualised CT-fostering Scratch project functionality rubrics (e.g., coding patterns and their underlying code constructs) or performance descriptions as indirect CT learning intentions and criteria.

Since programming is a demonstration of CT [23], the contents that students can implement in Scratch projects as summarised in Table 1 can be elicited as evidence of their CT (2). However, programming projects are not direct measurements of thinking, and there has been justified questioning concerning students' learning of computational concepts while working with such tools as Scratch [47]. However, signs of validity in assessing CT in the context of programming have begun appearing [48]. The examination of code constructs within semantically meaningful coding patterns could further improve the validity of the assessment [50]. Comprehensive rubrics for such contents could be adopted in future empirical research assessing students' CT in a wide-ranging and systematic manner attempting to, for instance, examine the issue of validity further, gain rich empirical insight, or weigh the usefulness of such rubrics in classroom practice.

It is crucial to complement the assessment by examining programming processes. [22] Prior studies examined students' programming activities via, for instance, observation, discourse analysis and interviewing (see Table 1). In schools, complicated research-designated tools are time-consuming. Additionally, prior studies assessed only certain CEPs and not CT comprehensively. Hence, an extensive and a pedagogically meaningful programming process assessment tool or rubric would also require further development. In future research, project content implementation could be examined alongside both peer-to-peer [29,32,52,57] and student-project [18,30] interactions. In-depth empirical examinations of interactions resulting in different kinds of contents could surface diverse desirable and undesirable programming activities. Such in-depth investigations

could also focus on discussing pedagogically meaningful assessment instruments for schools.

Lastly, the instantiation of CT-fostering contents could be supported in real time by providing targeted timely feedback for specific code segments in the students' projects (3) [34]. Although the feedback can be generated by teachers or peers, existing automated assessment tools (e.g., Dr. Scratch [44], NCV [46], Scrape [30]) that cover some areas of CT could be revisited to better satisfy this need.

5.3 | Fostering CT beyond the rubrics

Some CEPs were not straightforwardly contextualised in Scratch. First, removing redundant or unnecessary steps in algorithms (Efficiency) was not assessed beyond examining unscripted blocks as shown by Wilson et al [60]. Similarly, project functionality in general may not alone ensure positive user experience or functionality under all circumstances (Efficiency). Second, finding and collecting data from various sources and multilayered datasets (Data) may be problematic to effectuate in Scratch because it is primarily a media design tool and not a general-purpose programming language [38]. However, the domain of simulation-genre projects and the use of a range of I/O devices could potentially provide opportunities for data collection [9,13]. Thirdly, it is essential for students to understand that computers, operating systems, applications and programming languages are high-level abstractions of computations occurring in circuits and wires, how various digital devices could be used as a computer and identify real-world applications of CT (Abstraction and Automation). These CEPs could be meaningfully explored and assessed in the contexts of other programming tools and environments that can promote engaging learning activities for novice programmers (e.g., Lego Mindstorms [20], the App Inventor [47]) throughout compulsory education.

Then again, some CEPs were not contextualised in an in-depth manner. For instance, designing projects with several scripts and sprites as examined by Funke et al [19] contextualises managing complexity (Abstraction), but this task is likely very multilayered [24]. Similarly, the CEPs in Patterns and Generalisation and Problem decomposition [24] likely involve intricate cognitive tasks when instantiating code constructs and coding patterns as examined by, for instance, Seiter and Foreman [50] and Grover and Basu [21]. Moreover, alternate approaches to solving problems and "out-of-the-box thinking" (Creativity) are vague ideas that may only hold meaning in practical educational contexts. Then again, making fair and honest judgements in complex situations that are not free of values and constraints (Testing and debugging) and analysing situations and checking facts to make and verify predictions, making decisions and reaching conclusions (Logic) are very broad ideas that could relate to nearly all aspects of computational problem-solving. Furthermore, as the CEPs and the programming contents contextualising them emerged from previous works in this nascent research area, there can be relevant CT beyond what is currently known.

6 | CONCLUSIONS

Building on our current understanding of the key skills and areas of understanding associated with CT—often represented as its core concepts and practices and atomised here concretely as CT's CEPs—this study placed a particular focus on CT in the context of Scratch in K–9 (primary education). We summarised "CT-fostering" Scratch programming contents and activities from 30 studies into operational rubrics for teaching, learning and assessment at the primary school level. The results are applicable in educational practice, but the rubrics can be developed in future investigations. That said, the rubrics should not be regarded as complete or all-inclusive as CT is a developing research topic. However, by shedding light into its CEPs fostered via Scratch we also managed to raise some important areas that would benefit from further investigations. Some dimensions in CT could be meaningfully examined through quantitative metrics (e.g., code construct segments), whereas others may be more qualitative in nature (e.g., creative expression). The next aspiration could be applying a learning progression taxonomy to the contents and activities systematically.

Moreover, methods of formative assessment for contents and activities were explored. With this study as a springboard, our next steps are to refine pedagogically meaningful ways to assess CT in students' Scratch projects and programming processes. Validated assessment frameworks could potentially be extended into automated, formative learning-support systems that students can benefit from when programming.

What still gravely requires attention in CT is the quality of understanding that students develop while programming. Additionally, as CT is an interdisciplinary collection of skills and knowledge, it can develop through various tasks in different kinds of problem-solving contexts. To unify theories in CT education, the contents and activities in other programming environments (e.g., robotics, digital game-play) and nonprogramming domains should be reviewed in a similar fashion. Operational methods of assessing CT similarly in different contexts could be used to tackle the notorious transfer problem.

ORCID

Janne Fagerlund http://orcid.org/0000-0002-0717-5562
Päivi Häkkinen https://orcid.org/0000-0001-6616-9114
Mikko Vesisenaho http://orcid.org/0000-0003-1160-139X
Jouni Viiri http://orcid.org/0000-0003-3353-6859

REFERENCES

1. A. Aho, *Computation and computational thinking*. Ubiquity. 2011, Article No. 1.
2. C. Angeli, J. Voogt, A. Fluck, M. Webb, M. Cox, J. Malyn-Smith and J. Zagami, *A K–6 Computational thinking curriculum framework: Implications for teacher knowledge*, Edu. Technol. Soc. **19** (2016), no. 3, 47–57.
3. V. Barr and C. Stephenson, *Bringing computational thinking to K–12: What is involved and what is the role of the computer science education community?* ACM Inroads **2** (2011), no. 1, 48–54.
4. M. Ben-Ari, *Constructivism in computer science education*, The 29th SIGCSE technical symposium on computer science education, ACM, New York, NY, 1998, pp. 257–261.
5. L. Benton, I. Kalas, P. Saunders, C. Hoyles and R. Noss, *Beyond jam sandwiches and cups of tea: An exploration of primary pupils' algorithm-evaluation strategies*, J. Comput. Assist. Learn. **34** (2017), 590–601.
6. D. Black and D. Wiliam, *Developing the theory of formative assessment*, Edu. Assessment, Evaluation Accountability **21** (2009), no. 1, 5–31.
7. I. Blau, O. Zuckerman, and A. Monroy-Hernández, *Children's participation in a media content creation community: Israeli learners in a Scratch programming environment*, Learning in the technological era (Y. Eshet-Alkalai, A. Caspi, S. Eden, N. Geri and Y. Yair, eds.), Open University of Israel, Raanana, Israel, 2009, pp. 65–72.
8. S. Bocconi, A Chioccariello and J. Earp, The Nordic approach to introducing computational thinking and programming in compulsory education. Report prepared for the Nordic@-BETT2018 Steering Group. https://doi.org/10.17471/54007
9. K. Brennan and M. Resnick, New frameworks for studying and assessing the development of computational thinking. Paper presented at the meeting of AERA 2012, Vancouver, BC. 2012.
10. Q. Burke, *The markings of a new pencil: Introducing programming-as-writing in the middle school classroom*, J. Media Literacy Edu. **4** (2012), no. 2, 121–135.
11. Z. Chang, Y. Sun, T.Y. Wu and M. Guizani, Scratch Analysis Tool (SAT): A Modern Scratch Project Analysis Tool based on ANTLR to Assess Computational Thinking Skills. 2018 14th International Wireless Communications & Mobile Computing Conference. 2018; pp. 950–955.
12. J. W. Creswell, *Qualitative inquiry & research design: Choosing among five approaches*, 3rd ed., SAGE publications, Thousand Oaks, CA, 2012.
13. A. Czismadia, P. Curzon, M. Dorling, S. Humphreys, T. Ng, C. Selby and J. Woollard, Computational thinking. A guide for teachers. 2015. available at https://community.computingatschool.org.uk/files/6695/original.pdf
14. P. Denning, *Remaining trouble spots with computational thinking*, Commun. ACM **60** (2017), no. 6, 33–39.
15. B. Ericson and T. McKlin, Effective and sustainable computing summer camps. The 43rd ACM technical symposium on Computer Science Education. New York, NY: ACM. 2012; pp. 289–394.
16. D. Franklin, P. Conrad, B. Boe, K. Nilsen, C. Hill, M. Len, . . . R. Waite, Assessment of computer science learning in a Scratch-based outreach program. The 44th ACM technical symposium on Computer science education. New York, NY: ACM. 2013; pp. 371–376.
17. D. Franklin, G. Skifstad, R. Rolock, I. Mehrotra, V. Ding, A. Hansen, . . . D. Harlow, Using upper-elementary student performance to understand conceptual sequencing in a blocks-based curriculum. The 2017 ACM SIGCSE Technical Symposium on Computer Science Education. New York, NY: ACM. 2017; pp. 231–236.
18. A. Funke and K. Geldreich, *Measurement and visualization of programming processes of primary school students in Scratch*, The 12th Workshop on Primary and Secondary Computing Education, ACM, New York, NY, 2017, pp. 101–102.
19. A. Funke, K. Geldreich, and P. Hubwieser, Analysis of Scratch projects of an introductory programming course for primary school students. Paper presented at the 2017 IEEE Global Engineering Education Conference (EDUCON), Athens, Greece. 2017.
20. B. Garneli, M. Giannakos and K. Chorianopoulos, Computing education in K–12 schools. A review of the literature. Paper presented at the 2015 IEEE Global Engineering Education Conference (EDUCON), Tallinn, Estonia. 2015.
21. S. Grover and S. Basu, Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. The 2017 ACM SIGCSE Technical Symposium on Computer Science Education. York, NY: ACM. 2017;267–272.
22. S. Grover, M. Bienkowski, S. Basu, M. Eagle, N. Diana and J. Stamper, A framework for hypothesis-driven approaches to support data-driven learning analytics in measuring computational thinking in block-based programming. The Seventh International Learning Analytics & Knowledge Conference. New York, NY: ACM. 2017, pp. 530–531.
23. S. Grover and R. Pea, *Computational thinking in K–12: A review of the state of the field*, Educational Researcher **42** (2013), no. 1, 38–43.
24. S. Grover and R. Pea, *Computational thinking: A competency whose time has come*, Computer science education: Perspectives on teaching and learning in school (S. Sentance, E. Barendsen, and C. Schulte, eds.), Bloomsbury Academic, London, 2018, pp. 19–37.
25. F. J. Gutierrez, J. Simmonds, N. Hitschfeld, C. Casanova, C. Sotomayor and V. Peña-Araya, Assessing software development skills among K–6 learners in a project-based workshop with scratch. 2018 ACM/IEEE 40th International Conference on Software Engineering: Software Engineering Education and Training. 2018, pp. 98–107.
26. N. R. Haddaway, P. Woodcock, B. Macura and A. Collins, *Making literature reviews more reliable through application of lessons from systematic reviews*, Conserv. Biol. **29** (2015), no. 6, 1596–1605.
27. F. Heintz, L. Mannila and T. Färnqvist, A review of models for introducing computational thinking, computer science and computing in K–12 education. 2016 IEEE Frontiers in Education Conference. IEEE. 2016, pp. 1–9.
28. T. C. Hsu, S. C. Chang and Y. T. Hung, *How to learn and how to teach computational thinking: Suggestions based on a review of the literature*, Comput. Edu. **126** (2018), 296–310.

29. M. Israel, Q. M. Wherfel, S. Shehab, E. A. Ramos, A. Metzger and G. C. Reese, *Assessing collaborative computing: Development of the collaborative-computing observation instrument (C-COI)*, Comput. Sci. Edu. **26** (2016), no. 2–3, 208–233.
30. F. Ke, *An implementation of design-based learning through creating educational computer games: A case study on mathematics learning during design and computing*, Comput. Edu. **73** (2014), 26–39.
31. C. Lewis, How programming environment shapes perception, learning and goals. The 41st ACM technical symposium on Computer science education. New York, NY: ACM. 2010, pp. 346–350.
32. C. Lewis and N. Shah, How equity and inequity can emerge in pair programming. The eleventh annual International Conference on International Computing Education Research. New York, NY: ACM. 2015, pp. 41–50.
33. K. Lonka, M. Kruskopf and L. Hietajärvi, *Competence 5: Information and communication technology (ICT).* Phenomenal learning from Finland (K. Lonka, ed.), Edita, Keuruu, Finland, 2018, pp. 129–150.
34. S. Y. Lye and J. H. L. Koh, *Review on teaching and learning of computational thinking through programming: What is next for K–12?* Comput. Human. Behav. **41** (2014), 51–61.
35. A. Mackey and S. M. Gass, *Second language research*, Methodology and design, Lawrence Erlbaum Associates, Mahwah, NJ, 2005.
36. B. Mako Hill, A. Monroy-Hernández and K. R. Olson, Responses to remixing on a social media sharing website. Paper presented at the Fourth International AAAI Conference on Weblogs and Social Media, Washington, D.C. 2010.
37. J. Maloney, K. Peppler, Y. B. Kafai, M. Resnick and N. Rusk, Programming by choice: Urban youth learning programming with Scratch. The 39th SIGCSE technical symposium on Computer science education. New York, NY: ACM. 2008, pp. 367–371.
38. J. Maloney, M. Resnick, N. Rusk, B. Silverman and E. Eastmond, *The Scratch programming language and environment*, ACM Trans. Comput. Edu. **10** (2010), no. 4, 1–15.
39. L. Mannila, V. Dagiene, B. Demo, N. Grgurina, C. Mirolo, L. Rolandsson and A. Settle, Computational thinking in K–9 education. Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference. New York, NY: ACM, 2014, pp. 1–29.
40. R. E. Mayer, *Should there be a three-strikes rule against pure discovery learning? The case for guided methods of instruction*, Am. Psychol. **59** (2004), no. 1, 14–19.
41. O. Meerbaum-Salant, M. Armoni and M. Ben-Ari, Habits of programming in Scratch. The 16th Annual Joint Conference on Innovation and Technology in Computer Science Education. New York, NY: ACM. 2011, pp. 168–172.
42. O. Meerbaum-Salant, M. Armoni and M. Ben-Ari, *Learning computer science concepts with Scratch*, Computer Sci. Educ. **23** (2013), no. 3, 239–264.
43. G. Michaelson, *Teaching programming with computational and informational thinking*, J. Pedagog. Dev. **5** (2015), no. 1, 51–66.
44. J. Moreno-León, G. Robles and M. Román-González, *Dr. Scratch: Automatic analysis of Scratch projects to assess and foster computational thinking*, Revista de Educación a Distancia **15** (2015), no. 46, 1–23.
45. J. Moreno-León, G. Robles and M. Román-González, Towards data-driven learning paths to develop computational thinking with Scratch. IEEE Transactions on Emerging Topics in Computing. 2017.
46. G. Ota, Y. Morimoto and H. Kato, Ninja code village for Scratch: Function samples/function analyser and automatic assessment of computational thinking concepts. Paper presented at the 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Cambridge, England. 2016.
47. S. J. Papadakis, M. Kalogiannakis, V. Orfanakis and N. Zaranis, *The appropriateness of scratch and app inventor as educational environments for teaching introductory programming in primary and secondary education*, Int.. J. Web-Based Learn. Teach. Technol. **12** (2017), no. 4, 58–77.
48. M. Román-González, J. Moreno-León and G. Robles, *Combining assessment tools for a comprehensive evaluation of computational thinking interventions*, Comput. Thinking Edu. (S. C. Kong and H. Abelson, eds.), Springer, Singapore, 2019, pp. 79–98.
49. L. Seiter, Using SOLO to classify the programming responses of primary grade students. The 46th ACM Technical Symposium on Computer Science Education. New York, NY: ACM. 2015, pp. 540–545.
50. L. Seiter and B. Foreman, Modeling the learning progressions of computational thinking of primary grade students. The 9th annual international ACM conference on International computing education research. New York, NY: ACM. 2013, pp. 59–66.
51. A. Settle and L. Perkovic, Computational Thinking across the Curriculum: A Conceptual Framework. 2010. Technical Reports, Paper 13, available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.910.8295&rep=rep1&type=pdf
52. N. Shah, C. Lewis and R. Caires, Analyzing equity in collaborative learning situations: A comparative case study in elementary computer science. The 11th International Conference of the Learning Sciences. Boulder, CO: International Society of the Learning Sciences. 2014, pp. 495–502.
53. V. J. Shute, C. Sun and J. Asbell-Clarke, *Demystifying computational thinking*, Edu. Res. Rev. **22** (2017), 142–158.
54. B. Standl, *Solving everyday challenges in a computational way of thinking*, Informatics in schools: Focus on learning programming. ISSEP 2017. Lecture Notes in Computer Science (V. Dagienė and A. Hellas, eds.), **10696**, Springer, Cham, 2017, pp. 180–191.
55. J. M. Sáez-López, M. Román-González and E. Vázquez-Cano, *Visual programming languages integrated across the curriculum in elementary school: A two year case study using "Scratch" in five schools*, Comput. Edu. **97** (2016), 129–141.
56. M. Tedre and P. Denning, The long quest for computational thinking. 16th Koli Calling International Conference on Computing Education Research. New York, NY: ACM; 2016, pp. 120–129.
57. J. Tsan, F. J. Rodríguez, K. E. Boyer and C. Lynch, "I think we should...": Analyzing elementary students' collaborative processes for giving and taking suggestions. The 49th ACM Technical Symposium on Computer Science Education. New York, NY: ACM. 2018, pp. 622–627.
58. J. Voogt, P. Fisser, J. Good, P. Mishra and A. Yadav, *Computational thinking in compulsory education: Towards an agenda for research and practice*, Educ. Information Technol. **20** (2015), no. 4, 715–728.
59. C. Wangenheim, J. C. R. Hauck, M. F. Demetrio, R. Pelle, N. Cruz Alves, H. Barbosa and L. F. Azevedo, *CodeMaster—Automatic assessment and grading of App Inventor and Snap! programs*, Informatics in Education **7** (2018), no. 1, 117–150.
60. A. Wilson, T. Hainey and TM. Connolly, 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 (ECGBL), Cork, Ireland. 2012.
61. J. M. Wing, *Computational thinking*, Commun. ACM **49** (2006), no. 3, 33–35.
62. J. M. Wing, A definition of computational thinking from Jeannette Wing, available at https://computinged.wordpress.com/2011/03/22/a-definition-of-computational-thinking-from-jeanette-wing/
63. I. Zur-Bargury, B. Pârv and D. Lanzberg, A nationwide exam as a tool for improving a new curriculum. The 18th ACM conference on Innovation and technology in computer science education. New York, NY: ACM. 2013, pp. 267–272.

AUTHOR BIOGRAPHIES

Janne Fagerlund 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 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.

Päivi 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).

Mikko Vesisenaho is a senior researcher at the Department of Teacher Education, University of Jyväskylä, with background in education, contextual design, and computer science education. His ambition is for innovative reforms for learning with technology.

Jouni Viiri is a Professor of science and mathematics education at the Department of Teacher Education, University of Jyväskylä. His research focuses on physics education, in particular, the use of models and representations in physics education, argumentation, and communication between teachers and students.

How to cite this article: Fagerlund J, Häkkinen P, Vesisenaho M, Viiri J. Computational thinking in programming with Scratch in primary schools: A systematic review. *Comput Appl Eng Educ.* 2021;29:12–28. https://doi.org/10.1002/cae.22255