

JYU DISSERTATIONS 438

---

**Janne Fagerlund**

# Teaching, Learning and Assessing Computational Thinking through Programming with Scratch in Primary Schools

---



UNIVERSITY OF JYVÄSKYLÄ  
FACULTY OF EDUCATION AND  
PSYCHOLOGY

JYU DISSERTATIONS 438

---

**Janne Fagerlund**

**Teaching, Learning and Assessing  
Computational Thinking through  
Programming with Scratch in  
Primary Schools**

Esitetään Jyväskylän yliopiston kasvatustieteiden ja psykologian tiedekunnan suostumuksella  
julkisesti tarkastettavaksi yliopiston päärakennuksen salissa C1  
marraskuun 5. päivänä 2021 kello 12.

Academic dissertation to be publicly discussed, by permission of  
the Faculty of Education and Psychology of the University of Jyväskylä,  
in the Main Building, auditorium C1 on November 5, 2021 at 12 o'clock noon.



JYVÄSKYLÄN YLIOPISTO  
UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2021

Editors

Pekka Mertala

Department of Teacher Education, University of Jyväskylä

Ville Korkiakangas

Open Science Centre, University of Jyväskylä

Cover photo by Janne Fagerlund.

Copyright © 2021, by University of Jyväskylä

ISBN 978-951-39-8882-1 (PDF)

URN:ISBN:978-951-39-8882-1

ISSN 2489-9003

Permanent link to this publication: <http://urn.fi/URN:ISBN:978-951-39-8882-1>

## ABSTRACT

Fagerlund, Janne

Teaching, Learning and Assessing Computational Thinking through Programming with Scratch in Primary Schools

Jyväskylä: University of Jyväskylä, 2021, 151 p.

(JYU Dissertations

ISSN 2489-9003; 438)

ISBN 978-951-39-8882-1 (PDF)

This doctoral thesis explores the teaching and learning of a competence referred to as ‘computational thinking’ (CT) in the context of Scratch—an especially popular programming environment intended for young learners—in primary school classrooms. CT is an emerging topic in compulsory education that despite age-old roots in the discipline of computing has only recently begun to mature and gain a foothold in school curricula worldwide. It can be perceived as a multifaceted competence that students can learn by programming in age-appropriate ways (e.g. game design, robotics). In practice, however, CT’s journey to arrive in schools has been challenging. From a theoretical viewpoint, the challenges include the lack of uniformly defined concrete educational goals for CT and research-based pedagogical models for teaching and learning CT through programming. Consequently, large-scale studies have revealed shortcomings in teachers’ emphasis on CT and programming education at the grass-roots level.

This study sheds light on the topic in four main ways. First, it specifies the educational goals of CT in the context of programming at the primary school level. Second, the study evaluates ways to assess students’ CT in Scratch. Third, it develops new methods for assessing students’ CT. Fourth, the study presents empirical evidence from a case study conducted at the 4<sup>th</sup> grade level. Acquired through artefact analysis and programming process analysis, the evidence encompasses findings regarding students’ conceptual and practical encounters with CT through creative pair programming with Scratch in authentic classrooms.

The main findings of the study include comprehensive rubrics for ‘CT-fostering’ programming contents that students can manipulate and programming activities they can carry out in Scratch. They also include research-based evidence for teaching and learning CT in Scratch in addition to methods of assessing CT in Scratch in primary school classrooms. The broader contributions of the thesis include a tactile, curriculum-oriented outline of what CT can mean for primary education, particularly through programming education. Additionally, the contributions encompass rich evidence-based insight concerning ways CT can be taught and learnt collaboratively in Scratch and how such assessment practices that can enhance students’ CT learning in classrooms (i.e. clarifying learning goals, evincing student understanding and providing feedback) can potentially be facilitated in the classroom.

Keywords: computational thinking, programming, Scratch, assessment, pair programming, primary education

## TIIVISTELMÄ (ABSTRACT IN FINNISH)

Fagerlund, Janne

Ohjelmoinnillisen ajattelun oppiminen, opettaminen ja arviointi Scratch-ohjelmoinnin kautta peruskouluasteella

Jyväskylä: Jyväskylän yliopisto, 2021, 151 s.

(JYU Dissertations

ISSN 2489-9003; 438)

ISBN 978-951-39-8882-1 (PDF)

Tässä väitöskirjassa tutkitaan ”ohjelmoinnilliseksi ajatteluksi” (eng. *computational thinking*) lanseeratun osaamiskokonaisuuden oppimista, opettamista ja arviointia erityisesti lapsille ja nuorille kohdistetun ja suositun Scratch-ohjelmoinnin kontekstissa peruskoulun luokkahuonetilanteissa. Ohjelmoinnillinen ajattelu on perusasteen koulutuksessa uusi aihepiiri, jonka tieteellinen kypsyminen ja käytännön jalkautuminen eri maiden opetussuunnitelmiin on vasta käynnistynyt viime vuosina huolimatta aihepiirin ikivanhoista kytkökäsitä tietojenkäsittelyn tieteenalaan. Ohjelmoinnillinen ajattelu voidaan tulkita monitahoisena osaamisena, jota oppilaat voivat oppia ohjelmoimalla ikätasolleen sopivalla tavalla (esim. peliohjelmointi, robotiikka). Ohjelmoinnillisen ajattelun taival koulumaailmaan on ollut kuitenkin käytännössä haastava. Teoreettisesta näkökulmasta aihepiiriä haastavat puutteet yhdenmukaisesti määritellyistä konkreettisista kasvatustavoitteista sekä tutkimusperustaisista pedagogisista malleista sen oppimiseen ja opettamiseen ohjelmoinnin kautta. Suurten otantojen tutkimukset ovat myös osoittaneet opettajien kamppailevan ohjelmoinnillisen ajattelun ja ohjelmoinnin opetuksen painotuksessa ruohonjuuritasolla.

Tämä tutkimus valaisee aihepiiriä neljällä keskeisellä tavalla. Ensin tutkimus tämentää ohjelmoinnillisen ajattelun kasvatustavoitteita ohjelmoinnin kontekstissa peruskouluasteella. Toiseksi tutkimus tarkastelee erilaisia menetelmiä ohjelmoinnillisen ajattelun oppimisen arviointiin Scratchissa. Kolmanneksi tutkimuksessa kehitetään uusia menetelmiä oppilaiden ohjelmoinnillisen ajattelun osaamiseen arviointiin. Neljänneksi tutkimuksessa esitellään empiiristä aineistoa neljännellä luokalla toteutetusta tapaustutkimuksesta. Aineistoa käsiteltiin oppilaiden Scratch-ohjelmointiprojektien ja -prosessien analyysien keinoin. Aineisto kuvastaa, kuinka oppilaat olivat tekemisissä ohjelmoinnillisen ajattelun tiedollisten ja taidollisten osa-alueiden kanssa käyttäessään Scratchia luovaan pariohjelmointiin autenttisissa luokkahuonetilanteissa.

Tutkimuksen päälöydöksiin kuuluvat kattavat koosteet ohjelmoinnillisen ajattelun oppimista tukevista ohjelmointisisällöistä ja -käytänteistä Scratch-ohjelmoinnin kontekstissa. Löydöksiin kuuluu myös empiiristä näyttöä ohjelmoinnillisen ajattelun oppimisesta ja opettamisesta Scratchin kautta peruskoulun neljännellä luokka-asteella sekä oppilaiden ohjelmoinnillisen ajattelun oppimisen arviointiin soveltuvia menetelmiä. Yleisellä tasolla tämän väitöskirjan edistysaskeleisiin kuuluu konkreettinen, opetussuunnitelmaan suuntautunut hahmotelma ohjelmoinnillisen ajattelun merkityksestä perusopetuksessa erityisesti ohjelmoinnin opetuksen kautta. Edistysaskeleisiin kuuluu lisäksi rikasta tutkimusperäistä tietoa ohjelmoinnillisen ajattelun opettamisesta ja yhteistoiminnallisesta oppimisesta luovan Scratch-ohjelmoinnin kautta sekä luokkahuonetilanteisiin tarkoitetuista arviointikäytänteistä, joilla voidaan teoriassa tukea oppilaiden ohjelmoinnillisen ajattelun oppimista (ts. oppimistavoitteiden kirkastaminen, osaamistason selvittäminen, palautteenanto).

Avainsanat: ohjelmoinnillinen ajattelu, ohjelmointi, Scratch, arviointi, pariohjelmointi, perusopetus

**Author's address**

Janne Fagerlund  
Department of Teacher Education  
University of Jyväskylä, Finland  
janne.fagerlund@jyu.fi

**Supervisors**

Professor Päivi Häkkinen  
Finnish Institute for Educational Research  
University of Jyväskylä, Finland

Adjunct Professor (docent) Mikko Vesisenaho  
Department of Teacher Education  
University of Jyväskylä, Finland

Professor (emeritus) Jouni Viiri  
Department of Teacher Education  
University of Jyväskylä, Finland

**Reviewers**

Professor Matti Tedre  
School of Computing  
University of Eastern Finland, Finland

University Lecturer Jari Laru  
Faculty of Education  
University of Oulu, Finland

**Opponent**

Professor Matti Tedre  
School of Computing  
University of Eastern Finland, Finland

## FOREWORD

This thesis turned out to epitomise a surprisingly memorable adventure. To begin giving forewords for it, I must start from way back.

When I was a kid in the 90s, my big brother showed me how to design my own computer programs, such as a simple virtual Christmas calendar, with a programming environment called Visual Basic on our home computer. A few years later, I was introduced to coding animations with ASCII characters (letters, number, symbols) in an elective QBasic course, which was the best course ever – thank you teacher Teuvo Kaipainen! In my late teens, I taught myself to code narrative gameplay scenarios for my friends to play in Warcraft III at our LAN parties. When I minored in Educational Technology at the university, an inspiration to code reignited, and among my pastimes was coding a digital tool for my wife to calculate her net salary based on her irregular work shifts. After having eventually began conducting research on programming education (i.e. completing this thesis), I received a few opportunities to assist my childhood friend Ville, a geology student, in computationally modelling something like ancient sedimental layers on Finnish lake bottoms with Microsoft Excel.

In short, I seem to have always taken joy in using computers for solving problems and being creative. Coding appears to have combined these tasks aptly for me. Despite me pursuing a degree in Education, it seemed only according to expectations that this affection snuck into my career somehow.

This thesis took its first solid steps when the new Finnish primary school core curriculum was coming into effect just before 2016. An activity called ‘computer programming’ was steering towards primary schools and the teaching practice of school teachers, who seemed mainly wary and insecure about the entire thing. Yet, I could not avoid reliving all the coding-related joy of my personal history. Lending a hand to this educational reform on a grander scale seemed like a quest tailored just for me, so I had to embark on it.

Acquiring a doctorate through this thesis turned out to be a rather unstable ride. It was not always easy to march on as a novice researcher, especially when working with a topic that was accompanied by strong societal expectations and even hype. The topic developed in a manic pace: 60% of the references cited in this thesis were published *while this study was in progress!* Yet, regardless of its hotness, the topic turned out to be very foreign and distant to so many especially in the field of Education where digital technology altogether struggles to find comfortable residency. To wit, I was required to be a ceaseless learner in every respect from the beginning to the end. I often felt like I was paving a lone path with roars of cheers by my side, but only a few signposts to show the way.

Despite everything, I always found some way ahead. Perhaps the most notable benefactor was indeed the topic itself; its ‘hotness’, as praised so often by fellow colleagues. Most critically, I could not have built this work without those who trustingly chose to fund my learning. I address the greatest expressions of gratitude to the Department of Teacher Education at the University of Jyväskylä, the Central Finland Regional Fund, the Emil Aaltonen Foundation, and the Ellen

and Artturi Nyysönen foundation for being convinced of the importance of learning to understand this topic better.

This thesis would not have sprung to life either without the individual people who participated in concretely steering it to its conclusion. Above all, I honorably acknowledge excellent reviewers Professor Matti Tedre and Dr. Jari Laru for the feedback to strengthen this work and the encouraging words illustrating where I seemed to have been especially successful. Thank you especially Matti Tedre for agreeing to be my opponent on the day of my defence. I also thank my supervisors, Professor Päivi Häkkinen, Dr. Mikko Vesisenaho, and Professor emeritus Jouni Viiri. It has been humbling to bask in your expertise and try to steal your time to pick up all the things that I possibly could during our shared time with this challenging project. With a pinch of bittersweetness, I also assert special recognition to the journal reviewers who shot my scholarly ideas in this unsettled topic down so many times, calibrating my understanding of how to do research in academia and challenging my resolve.

Perhaps my most heartfelt gratitude goes to the three lovely teachers who eagerly volunteered to participate in the data collection of this study. Most of all, gratitude goes to their wonderful students who were astonishing in their creativity and excitement amidst the coding activities I got to introduce to them. I especially remember that one 4<sup>th</sup> grade girl who appeared to discover that same spark for coding within her as I did in my youth. Aside the scientific impact this study may make, the enthusiasm awoken in those children alone felt like I have achieved something good and important.

This thesis had always intended to be of pragmatic value; an enterprise to be reflected fundamentally on societal reality and the educational practice of teachers and schools. With that in mind, maturing the reasoning in this work was thankfully in no manner done in isolation. A monumental token of gratitude goes to the Innokas Network with all the exceptionally excited teachers from all around the country and beyond. Thank you especially, Dr. Tiina Korhonen, for inviting me to work as a regional coordinator in Innokas, for guiding my way to other great opportunities, such as working as a leading teacher for the EU Code Week, and for inspiring the whole gang of Innokas educators onward in being collaborative and innovative with educational technology... without ever forgetting to enjoy the ride!

Other very important people also nudged my understanding forward amidst this adventure. Warm individual recognitions go to Professor Mirja Tarnanen, Assistant Professor Pekka Mertala, Senior Researcher Kaisa Leino, and Jukka Lehtoranta – sterling researchers and educators with whom I had the luck to bounce ideas and make many contributions in the variety of projects we had and have going on. I also acknowledge the international computing education research community with which I got to make acquaintance in the conferences and doctoral consortiums. Thanks to our digi-pedagogically oriented JYULED team for all the cooperation. Acknowledgement goes especially to Memma Juntunen for your compassionate support and collegiality toward me, especially



when I started working at the university. A sincere thank you goes to Emilia Ahlström who helped me a lot with Article III. I also thank Sini Salmela who assisted me in collecting the data while completing her own master's thesis.

Having reached this intermediate point in my academic journey, I find an opportunity to profess appreciation of the more affectionate kind. I earnestly bow to our informal postgraduate students' peer support group (originally known as 'The Almost Dead Postgrads' Society') that helped me especially when I needed more will and confidence for enduring the doctoral turbulence we all learnt to know so well. In particular, I graciously mention (soon-to-be Dr.) Anne Martin – with whom I found myself to share so many qualities yet least in the topics of our theses – who braved her own doctoral gauntlet concurrently and accounted for much peer support. Thank you also all the people in our Ruusupuisto office, the notorious 2D, who contributed in making our shared workdays funky and interesting. Thank you, all the meaningful people at our #JYUnique university and beyond who I fail to mention by name or affiliation.

I suppose that it was my big brother Tomi, coincidentally a professional programmer, who opened my path to the topic of this thesis when we were kids. Despite being frustrated at the whiny little brat, you showed me how to code, and I used to copy your code when I could not design all that fancy stuff myself. You did not always like that, which is why sometimes mom, dad, or our big sister had to try to settle our disputes. To my family and all my friends: thank you for the shared life during the completion of this work.

Lastly and most lovingly, I thank you, Suvi, my best friend and wife. You were there with me during the mundane and marvelous moments alike. With you I got to see our precious daughters Stella and Malla coalesce from stardust through ancient magic. You three give me the reason.

This thesis is dedicated to my firstborn daughter, Stella. She successfully programmed a Bee-bot robot as a 1½-year-old when she 'co-trained' teachers with me in a programming-themed workshop at the University of Helsinki. More recently she has begun to provide me creative ideas to design in our own Scratch games (in the cover photo). She is truly a next-generation computational thinker.

Jyväskylä 27.9.2021

Janne Fagerlund

## AUTHOR'S CONTRIBUTION

*Design of the study and methods.* The author designed the programming course with the co-authors' advice and was responsible for its implementation with instructional assistance from Sini Salmela and the participating teachers. The author chose the methods of data collection and the analysis methods in each article with the co-authors' advice.

*Data collection.* The author was responsible for participant selection and data collection, with technical assistance from Sini Salmela.

*Data analyses.* The author developed the analysis rubrics and coded all data. Blind coding was performed in Article III by Emilia Ahlström.

*Findings and writing.* The author of this thesis wrote all three original research articles as the first author, with comments by the co-authors.

## LIST OF ORIGINAL ARTICLES

- I Fagerlund, J., Häkkinen, P., Vesisenaho, M., & Viiri, J. (2021). Computational thinking in programming with Scratch in primary schools: A systematic review. *Computer Applications in Engineering Education*, 29(1), 12–28. <https://doi.org/10.1002/cae.22255>
- II Fagerlund, J., Häkkinen, P., Vesisenaho, M., & Viiri, J. (2020). Assessing 4th grade students' computational thinking through Scratch programming projects. *Informatics in Education*, 19(4), 611–640. <https://doi.org/10.15388/infedu.2020.27>
- III Fagerlund, J., Vesisenaho, M., & Häkkinen, P. (under review). Fourth grade students' computational thinking in pair programming with Scratch: A holistic case analysis

## FIGURES

Figure 1.	Publication years of works cited in this thesis .....	20
Figure 2.	The relationship of key concepts in this thesis (modified from Zhang & Nouri, 2019).....	25
Figure 3.	The actions taken in this thesis through the three research articles ..	27
Figure 4.	An illustration of the Logo programming environment primarily used in the 1970s .....	30
Figure 6.	Key concepts and practices that prior literature associates with CT .....	37
Figure 7.	Some contemporary programming environments utilised in schools.....	42
Figure 8.	Similar programs in two programming languages (left: Scratch, right: Java-Script in Micro:bit) .....	44
Figure 9.	Sample screenshots from Scratch .....	46
Figure 10.	The relationship of CT and programming as interpreted in this study .....	55
Figure 11.	The data collection setting and sample footage for Article III .....	67
Figure 12.	The mixed methods approach of analysing the data for the two RQs .....	68
Figure 13.	Sample CT-fostering Scratch programming contents.....	92
Figure 14.	A general learning path for CT-fostering contents in creative coding with Scratch .....	93
Figure 15.	A proposed model for learning CT through manipulating programming contents in Scratch .....	95
Figure 16.	Ways of providing spontaneous feedback for students programming in Scratch.....	102
Figure 17.	Hypothetical feedback for CT-fostering programming contents in Scratch projects.....	104
Figure 18.	Scratch programming during GameDev, a game design competition organised at the Innokas programming and robotics tournament (photo: Miika Miinin) .....	118

## TABLES

Table 1.	Scratch projects made weekly during the course.....	66
Table 2.	Overview of the articles.....	71
Table 3.	An incipient synthesis of profitable actions in (i.e. potentially good ways for) carrying out CT-fostering programming activities in Scratch .....	96
Table 4.	Essential target areas to evince students' introductory skills and understanding in CT through Scratch at the primary school level ...	99

# CONTENTS

ABSTRACT	
TIIVISTELMÄ (ABSTRACT IN FINNISH)	
FOREWORD	
AUTHOR'S CONTRIBUTION	
LIST OF ORIGINAL ARTICLES	
FIGURES AND TABLES	
CONTENTS	

1	INTRODUCTION .....	15
1.1	A computer revolution.....	15
1.2	Programming returns to schools .....	17
1.3	Computational thinking – an emerging competence.....	19
1.4	Call for research in CT through programming .....	23
1.5	The focus of this study.....	24
2	THEORETICAL BACKGROUND .....	28
2.1	Computational thinking and programming.....	28
2.1.1	A historical overview .....	29
2.1.2	Defining computational thinking.....	31
2.1.3	The relationship of CT and programming.....	34
2.1.4	CT's core educational principles .....	36
2.2	Teaching and learning CT in Scratch .....	40
2.2.1	Scratch amid contemporary programming environments.....	41
2.2.2	Creative computing with Scratch.....	45
2.2.3	Pedagogical underpinnings in graphical programming .....	47
2.2.4	Assessment for learning.....	51
2.2.5	Assessing programming contents and activities.....	53
3	AIMS AND RESEARCH QUESTIONS .....	58
4	CONTEXT AND DESIGN.....	60
4.1	Primary education in Finland .....	60
4.1.1	The core curriculum .....	60
4.1.2	ICT competence (T5).....	61
4.1.3	Programming in Finnish schools.....	62
4.2	Participants.....	63
4.3	Data collection.....	65
4.4	Data analysis .....	67
4.5	Research philosophy.....	68
4.6	Ethical aspects .....	69

5	SUMMARY OF ARTICLES .....	71
5.1	Article I: Computational thinking in programming with Scratch in primary schools: A systematic review .....	71
5.1.1	Aims.....	71
5.1.2	Methods.....	72
5.1.3	Main results.....	72
5.1.4	Discussion.....	74
5.2	Article II: Assessing 4 <sup>th</sup> grade students' computational thinking through Scratch programming projects.....	75
5.2.1	Aims.....	75
5.2.2	Methods.....	76
5.2.3	Main results.....	76
5.2.4	Discussion.....	78
5.3	Article III: Fourth grade students' computational thinking in pair programming with Scratch: A holistic case analysis .....	80
5.3.1	Aims.....	80
5.3.2	Methods.....	80
5.3.3	Main results.....	81
5.3.4	Discussion.....	83
6	CONCLUSIONS.....	85
6.1	Summary of key findings .....	85
6.2	Contributions of the study .....	86
6.2.1	CT in the curriculum.....	87
6.2.2	Formative assessment of CT in Scratch .....	91
6.2.2.1	Clarifying learning goals.....	91
6.2.2.2	Evincing student understanding.....	96
6.2.2.3	Providing feedback.....	99
6.3	Limitations of the study .....	105
6.3.1	Investigating CT.....	105
6.3.2	Case study with Scratch.....	108
6.3.3	Observational methods .....	109
6.4	Future research.....	111
6.5	Closing remarks .....	114
	YHTEENVETO.....	119
	REFERENCES.....	125
	APPENDICES.....	150
	ORIGINAL ARTICLES	

# 1 INTRODUCTION

'It's more fun to compute.' -Kraftwerk, 1981

This thesis concerns the teaching and learning of an emerging, multifaceted competence called computational thinking at the primary school level. This competence is investigated in the practical context of computer programming (colloquially called coding) in 4<sup>th</sup> grade students' classroom practice while using a very popular programming environment called Scratch.

This topic is relatively novel in compulsory education and not well understood from the viewpoint of scientific research, and educational experts' conceptions and expectations concerning this educational reform have varied. Therefore, to bring much-needed clarity to the topic and ground the choices made in this thesis, this chapter provides a comparatively introspective and rich introduction. In particular, the chapter elucidates the societal changes that have brought programming and computational thinking to compulsory education across the world (in sections 1.1 and 1.2), discusses how these topics have been rationalised theoretically and practically in different ways (in section 1.3), describes focal gaps in previous research (in section 1.4) and outlines the scholarly development of the thesis (in section 1.5).

## 1.1 A computer revolution

We live in a highly digitalised society. Increasing in quantity and complexity, the technological dimensions of everyday life include such relatively familiar phenomena as computers, the Internet and robotic systems but also more abstract ones, such as the Internet of Things, artificial intelligence and big data (Korhonen, 2017). Most people in the digitally developed world have a personal mobile device they can use for daily activities, such as purchasing goods from abroad, sharing media on virtual social networks and optimising navigational routes while traveling. Mobile devices are, in fact, computing machines (colloquially

called computers), just like traditional personal desktop computers (PCs) or laptops but smaller. However, today's smartphones have more processing power than, for example, the computer used in sending Neil Armstrong to the moon in 1969.

Although it has been known for a long time that there are certain types of tasks computers cannot carry out (Turing, 1937), the limitations of what computers can do seem to be becoming ever fewer. Today, artificially intelligent systems can automatically detect cancer growth on mammograms. Machine learning ensures that consumers on the web are offered recommended products for shopping. Engineers can test passenger flights' safety issues with computer simulations. Chemists can use algorithms to identify chemicals to improve reaction conditions to improve yields. Educators can use programmed cognitive tutors and adaptive learning environments to personalise students' learning trajectories and improve learning outcomes (Buitrago Flórez et al., 2017; Grover, 2018). Computers were recently used to employ algorithms to help seek a cure for the ongoing global pandemic<sup>1</sup>. Stephen Wolfram even highlighted<sup>2</sup> the potential of attaining a 'theory of everything' in physics via computational methods.

On the flip side, computers have brought rather worrying concerns, such as job loss due to automation, mass surveillance, cyber war and sales of personal data (Denning & Tedre, 2019). Social media services employ algorithms that display selected content to their users and may amplify feelings of inferiority and cause mental problems (Boers et al., 2019). The World Health Organization has included digital game-playing as a disorder in the listing of standard diseases<sup>3</sup>. With constantly developing technology in a world where the role of creative problem solving is predicted to increase as machines take over routine tasks in various fields, new challenges arise in human thinking, emotions and ethics. As society digitalises at an accelerating pace, computational competence is vital for all people to ensure that humankind steers towards a productive and responsible future (Lonka et al., 2018). Kafai and Burke (2013a) raise an interesting question: Do the 'digital natives' have sufficient capacity to wield digital technology in a critical, creative and selective manner? In short, the scientific world and different sectors of industry are filled with complex problems that can greatly benefit from the innovative solutions of capable humans who have an understanding of a particular discipline and of computers (Grover, 2018; Martin, 2018).

Ultimately, all computers follow the same technological baseline rules that were developed by humans for humans to overcome their own slowness and errors when performing rote tasks that could be *computed* effectively with automated devices. In short, computing is 'any goal-oriented activity requiring, benefiting from or creating computers', and it essentially includes 'processing, structuring and managing various kinds of information' (Association for

---

<sup>1</sup> <https://onezero.medium.com/computer-scientists-are-building-algorithms-to-tackle-covid-19-f4ec40acdba0>

<sup>2</sup> <https://writings.stephenwolfram.com/2020/04/finally-we-may-have-a-path-to-the-fundamental-theory-of-physics-and-its-beautiful/>

<sup>3</sup> <https://www.who.int/news-room/q-a-detail/addictive-behaviours-gaming-disorder>



Computing Machinery, 2005, p. 9). The information that computers can process, *data*, can be represented as seemingly infinite series of binary digits (1s and 0s), which are merely abstractions of physical voltage states and their changes in microchips. Today's computer applications operate on much higher levels of abstraction; perhaps familiar examples are programming languages that more or less represent human language, although computer users can now largely disregard programming languages entirely and focus on using software (or 'apps') (Denning & Tedre, 2019).

Computing has existed as a human experience for a long time. Procedures similar to algorithms and information representation with numbers and symbols date back thousands of years. For example, the abacus was used to perform computational tasks in ancient Babylon around 1800 BC. Theoretically, a human being could perform the same computational tasks as an industrial computer used with, for example, particle accelerators, albeit much more slowly and most likely with a plethora of errors. The formalisation of key computational concepts in the 1930s and the rapid development of microchip technology in the 1990s enabled computational tasks to be carried out more quickly and efficiently with digital computers. Computing also expanded to different branches of science as a new meaningful way of doing scientific research. It was adopted primarily in studying natural phenomena by modeling them as information processes and using computing to understand them. However, such activities required new kinds of intellectual resources from people thinking of ways to implement such processes (Denning & Tedre, 2019; Tedre, 2015; Tedre & Denning, 2016). In other words, a major deficiency with the digital devices, gadgets or machines that do computing is that no matter how fast or accurate they are when doing mechanical calculations, they are not creative and require human help to solve complex real-life problems. They need to be told intelligently beforehand how they should process particular data to reach a desired outcome. They need a declaration of preset instructions, *a computer program*, that includes sequences of computational steps or *algorithms* (Van Roy & Haridi, 2003).

## 1.2 Programming returns to schools

The ramifications of a digitally developing society has been a hot educational topic in recent decades. Much discourse has focused on how to use technology meaningfully to *support learning processes* in addition to what technological topics are justifiable *targets of learning*. More or less encompassing both viewpoints, information and communications technology (ICT) competence is an integral sub-topic of the so-called 21<sup>st</sup> century skills (Binkley et al., 2012). However, as Giordano et al. (2015) poetically put it in describing the current state of this educational topic, 'Out goes ICT and how to use Microsoft Office; in comes coding and computer science.'

Themes such as robotics, programming, informatics, computer science (CS) and computing have begun to make stronger inroads into educational systems

worldwide, perhaps more than ever before. For example, in 2014, England adopted computing as mandatory content for all primary school (i.e. grades 1–9) students. In 2016, Finland was among the frontrunners in adopting computer programming, especially ‘graphical programming’ (see details below), in the national primary school core curriculum as mandatory learning content for all students (Opetushallitus, 2014). More than 20 countries have subsequently followed suit, and the numbers seem to be growing (Balanskat et al., 2017; Bocconi et al., 2018; Heintz et al., 2015; Mannila et al., 2014). In fact, according to a relatively recent international survey, topics such as artificial intelligence, cybersecurity, machine learning, robotics and web systems are included in several countries’ intended and enacted curricula from pre-primary to senior secondary years (Falkner et al., 2019).

Age-appropriate learning activities amidst such topics in schools have often encompassed programming, particularly ‘graphical’ (or ‘block-based’) programming (Grover & Pea, 2013). Industrial manufacturers have made available several computational kits, including physical kits with and without electronics, virtual kits and hybrid kits with virtual or tangible programming blocks. In fact, the popular ‘unplugged’ movement provides a myriad of learning resources that do not require the use of digital devices, which has made programming education available even for pre-schoolers and students who do not have access to technology (Brackmann et al., 2017; 2019; Looi et al., 2018; Moschella, 2019; Wu et al., 2018; Yu & Roque, 2019). Several previous studies have also compared programming environments in terms of their appeal and effect in learning (Szabo et al., 2019).

Among the most popular and evidently appealing graphical programming environments adopted in primary schools is *Scratch* (Garneli et al., 2015; Lye & Koh, 2014; Szabo et al., 2019). Scratch can be used with a web browser for free to design interactive media, such as games, stories and animations, and share them with fellow designers across the world. The media projects are designed by programming, that is, designing step-by-step sequences of instructions (algorithms) with pre-set code blocks (Resnick et al., 2009).

The presence of computing, programming or a similar topic<sup>4</sup> in primary education is by no means unprecedented (Denning & Tedre, 2019). Kafai and Burke (2013a) published an aptly titled paper, *Computer programming goes back to school*, and pointed out that coding has been introduced in schools in history. Particularly during the turning point of the 1980s–1990s, there was excessive enthusiasm for programming in learning. However, this enthusiasm greatly diminished by the mid-1990s due to a lack of meaningful subject-matter integration (i.e. teachers’ feeling that it was unnecessary), lack of qualified instruction and the emergence of modern multimedia technologies (e.g. CD-ROMs) and other digital novelties that were of interest to educators (e.g. teaching students to surf the Internet). Why should students learn this clumsy and seemingly not very important skill (Kafai

---

<sup>4</sup> The words ‘programming’ and ‘coding’ are occasionally used in various texts as interchangeable synonyms (e.g. Balanskat & Engelhardt, 2015). Coding (i.e. writing computer code) may be an attractive term in vernacular languages (Zhang & Nouri, 2019), but it is actually only a part of programming (i.e. program construction) let alone computational thinking (a foundation of thought) (Tedre & Denning, 2016).

& Burke, 2013a)? Today, educational systems across the world seem convinced that they should. By fairly common consent, recent discussions have coalesced to accommodate one particular emergent term to this frontier: *computational thinking* (CT)—a fundamental competence that everyone in the world should acquire (Wing, 2006; 2008).

### 1.3 Computational thinking – an emerging competence

CT erupted definitively into public view in 2006. Jeannette Wing, a professor of CS at Carnegie Mellon University, states that CT is a fundamental skill for everyone and is used everywhere in today’s digital world. It is a skill that school children should acquire, just as they learn reading, writing and math skills (Wing, 2006). Wing’s timely call for action ignited a surge of discussions, scientific studies and educational initiatives on learning computer programming in schools (Tedre & Denning, 2016). Today, many educational institutions are interested in incorporating CT or proximal topics, such as computing, CS or programming, into school curricula (Balanskat & Engelhardt, 2015; Heintz et al., 2015; Mannila et al., 2014). CT-related initiatives have even been recently seen at the preschool educational level (e.g. Bers et al., 2019).

CT is a comparatively new, continuously maturing and trending educational topic; the amount of published research on the topic nearly quintupled between 2015 and 2017 (Hsu et al., 2018). In particular, the number of studies conducted at the compulsory school level has increased rapidly in recent years. This is evidenced even by the publication years in the reference list of this thesis (Figure 1), which exemplifies the fast pace of growing knowledge on the topic and the necessity of being aware of ongoing developments. The first distinctly CT-themed international scientific conference, CTE2017, was organised in Hong Kong in 2017 (Kong et al., 2017). According to a recent scientometric study of CT (Saqr et al., 2021), ‘CT research has been US-centric from the start, and continues to be dominated by US researchers both in volume and impact’. Approximate milestones in the development of the topic could be roughly categorised as follows: (1) early research in computing education (e.g. Papert, 1980; Resnick et al., 1988), (2) the emergence and first definitions of CT (e.g. Wing, 2006; Lee et al., 2011), (3) descriptions of the characteristics of CT (e.g. Barr & Stephenson, 2011; Shute et al., 2017) and (4) the rapid increase in empirical studies and reviews on CT (e.g. Moreno-León et al., 2015; Zhang & Nouri, 2019).

The youth of CT as a research topic is also evidenced by the lack of well-established theories, terminology and, especially, ways to interpret the educational relevance of CT. In fact, several motifs have prompted educational initiatives to include CT in school curricula. First, reacting to the past and the current digitalisation of the world and acting proactively for the foreseeable future, it is necessary to increase enrolment in the science, technology, engineering and mathematics (STEM) disciplines at the university level (see e.g. Trilles & Granel, 2019).

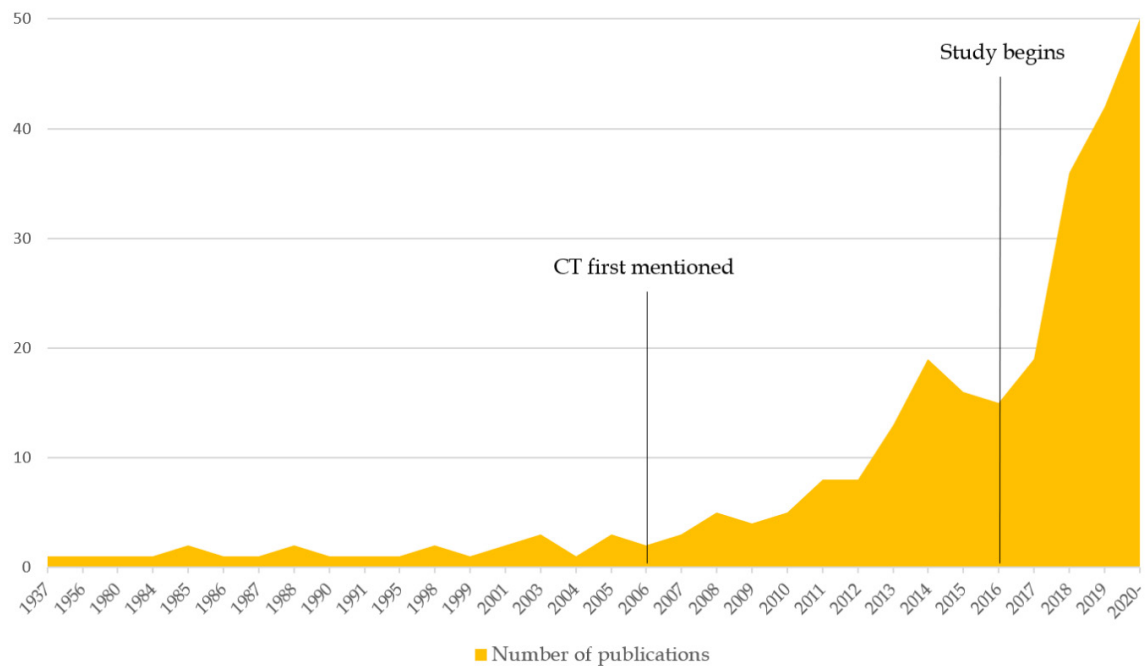


Figure 1. Publication years of works cited in this thesis

2020) and to train more qualified workers for the ICT sector<sup>5</sup>. Therefore, a key motif for introducing CT through coding in schools is to trigger students' interest in computer programming and provide them opportunities to become more interested in it for further training and working as professional coders.

A second motif for introducing CT through programming in education steps away from technology and the discrete application of computational tools and methods. It emphasises that CT promotes general and transferable thinking skills that can be utilised in a variety of problem-solving situations, such as in studying, working or carrying out everyday procedural tasks in non-computational settings. Although this claim has been shown to be little else than a myth for the lack of scientific evidence (De Bruyckere et al., 2020, p. 7–10; Denning 2017), nationally famous teacher training kits are among famous examples making such promises in a rather outspoken manner: 'programming helps learn generally useful cognitive skills' (Liukas & Mykkänen, 2014).

The inclusion of CT-related topics in curricula has been thirdly motivated by the ability of CT to promote an understanding in computing that is relevant in all work sectors. Conrad Wolfram argued<sup>6</sup>, perhaps slightly provocatively, that schools should cease introducing calculation because computers do it much more efficiently: 'humans should learn to use computing tools to address increasingly complex problems' by defining questions, abstracting them to computable

<sup>5</sup> [https://minedu.fi/artikkeli/-/asset\\_publisher/kiuru-ohjelmointi-peruskoulun-ope-tussuunnitelman-perusteisiin](https://minedu.fi/artikkeli/-/asset_publisher/kiuru-ohjelmointi-peruskoulun-ope-tussuunnitelman-perusteisiin)

<sup>6</sup> <https://www.forbes.com/sites/tomvanderark/2020/06/29/stop-calculating-and-start-teaching-computational-thinking/#1a13eb583786>

forms, computing answers and interpreting the results. In essence, CT allows understanding how computing can be utilised in different subject areas. It can thus be seen to provide productive skills akin to writing even if students do not become professional programmers. In a sense, everyone could learn to do 'CT jobs' in their work (Denning & Tedre, 2019; Grover & Pea, 2018). Perhaps the key question is: Can you manage without coding in future workplaces? Although the future is difficult to predict, there is a revolutionary prospect in this idea: people working in different fields could find entirely new and transformative solutions to problems that were not necessarily discovered before examining the range of problems in that field through 'CT goggles' (Tedre & Denning, 2016).

As a consequence of the above motifs for introducing CT in schools, the majority of the nascent research on the topic has situated within the tradition on educational psychology. In particular, studies have adopted a cognitive viewpoint, that is, that of individual learners, adhering to students' understanding of CT as a conceptual whole and their abilities to solve different kinds of problems with CT (Kafai et al., 2019). To that end, CT and programming have been associated with the tradition of crafts pedagogy – hands-on doing and solving problems by manufacturing concrete artefacts for pragmatic utilisation (e.g. Blikstein, 2020).

However, when enacting programming education care must be taken to avoid falling into a discourse in which children are regarded merely as a tool for industry. CT can also be (and has been more rarely) examined from the viewpoint of educational sociology, which Kafai et al. (2019) concretise as a 'situated viewpoint' (i.e. that of, for example, communities of practice) and a 'critical viewpoint' (i.e. that of society, structures of power, etc.). Programming should not be seen to present merely logical exercises and functional solutions that are free from values. Instead, computer code acts as someone's choice – a socio-material text, laden with values, worldviews, identities and aims that can influence, control or manipulate societally or socially (Mertala et al., 2020). Accordingly, CT and programming can also be associated with a somewhat newer paradigm, something akin to 'computational literacy'. CT can aim to cultivate creativity, participation in social communities and gaining a critical view and an active role regarding how computing appears in the manifold social, political, cultural and ethical dimensions in the world (see Bocconi et al., 2018; Kafai et al., 2019; Lonka et al., 2018; Williamson, 2016).

Through the viewpoint of CT as a type of literacy, CT competence can also be perceived as an increasingly important component in 21<sup>st</sup> century education that manifests as a kind of digital 'citizenship' or 'agency' or perhaps a kind of computational 'awareness', 'sophistication', 'fluency' or 'wisdom'. In an emancipatory sense, it is justifiable to teach all students to understand something about the ways in which computing is shaping our shared environments and provide them with tools to safely and responsibly navigate life in a continuously computationalised society (Høholt et al., 2021; Lonka et al., 2018). As a related concept, Dufva and Dufva (2018) conceptualise 'digi-grasping': not just 'being in the dig-

ital or the use of the digital’ but ‘grasping – an embodied understanding and empowered agency – of digital phenomena’. Høholt et al. (2021) recently proposed a model for progression in ‘computational empowerment’, which encompasses a notion of ‘reflexivity regarding the effect of technology in one’s own life and in society’ whilst becoming proficient in the more problem-solving domain in CT. As an example, the Korean curriculum has included CT and ‘informational ethics’ for over a decade (Jun et al., 2014). Meanwhile, the role of such notions in the Finnish national basic education core curriculum and in teacher training has been nearly non-existent or at least very limited (Mertala et al., 2020).

With this rather diverse and unsettled background, CT continues its entry into primary education systems across the world, especially through programming activities. Although the scientific and pragmatic efforts surrounding CT education have perceived the term in different ways, they have resulted in particular definitive ideas regarding competence, which are also adopted as the underpinnings of CT in this study. In particular, CT is perceived to be applied by interpreting information processes in the world and designing computations while solving problems (Denning & Tedre, 2019). It is perceived to involve understanding, for instance, the concept of *algorithms*, which are sets of instructions that can be carried out to perform a task that solves a specific problem or a class of problems. Acquiring skills in and understanding *abstraction* are also required to represent the problems and their solutions with the tools, languages and symbols of computing. The above terms are among those that are typically referred to as the ‘key concepts and practices’ of CT, which include several additional ones in somewhat disparate categorisations (see e.g. Barr & Stephenson, 2011; Bocconi et al., 2018; Csizmadia et al., 2015; Grover & Pea, 2018; Shute et al., 2017). Additionally, focusing perhaps more on the attitudinal, dispositional or perceptive rather than the skill-related dimension in CT, such ideas as self-expression, questioning, perseverance, cybersafety, sustainability and computational ethics are held to be essential in CT (Barr & Stephenson, 2012; Brennan & Resnick, 2012; Duncan & Bell, 2015; Lonka et al., 2018).

Although there is no definitive answer yet as to what ‘CT-like’ ways to think and perform tasks all students should learn, various tools and pedagogical methods have been developed to assist teachers to introduce and students to become familiar with aspects of CT in practice. Hoppe and Werneburg (2019) state that CT becomes meaningful ‘in the creation of ‘logical artifacts’ that externalise and reify human ideas in a form that can be interpreted and ‘run’ on computers’. In other words, the naturalistic purpose of CT is to understand and create usable technological solutions (e.g. devices, gadgets, programs or information systems) (Connor et al., 2017; Denning & Tedre, 2019). Programming a digital computer to do a particular task is consequently regarded as a central pathway for developing in CT (Grover & Pea, 2013; 2018). As stated previously, such programming contexts as robotics, unplugged exercises and Scratch have been commonly employed in schools in the expectation that they foster CT in some meaningful way.

## 1.4 Call for research in CT through programming

Despite the strong interest in and multitude of tools and approaches for its implementation, the journey via which CT has come to schools has been difficult. One year after the implementation of the new curriculum in Finland, only slightly more than a fifth of Finnish primary and secondary school teachers had even tried programming with their students. Moreover, nearly 70% of teachers stated that they did not possess the required skills to use a graphical programming environment. Meanwhile, primary school students' programming skills were confirmed to be at an extremely low level (Kaarakainen et al., 2017).

Telling a similar story in 2019, the International Computer and Information Literacy Study (ICILS) conducted in 14 countries showed that there is much variety in 8<sup>th</sup> grade students' CT skills (Fraillon et al., 2020). ICILS also revealed that in Finland teachers generally emphasised the teaching of CT skills less in their teaching practice than teachers in other participating countries (Leino et al., 2019).

Most school teachers may lack the necessary skills to combine sufficient technological, pedagogical and content-related knowledge in teaching CT and programming and to integrate them with other subject matter (Bull et al., 2020). New necessary CT-related knowledge (e.g. pedagogical content knowledge) required of teachers has been rightfully problematised and the importance of teacher training highlighted by several researchers (e.g. Armoni, 2019; Hubbard, 2018; Iwata et al., 2020; Kong et al., 2020; Lamprou & Repenning, 2018; Mannila et al., 2018; Pears et al., 2019; Waite et al., 2020; Yadav et al., 2016). For instance, Rich et al. (2021) have found that teachers lacked confidence in teaching specific programming contents and CT concepts. Mäkitalo et al. (2019) have proposed a framework called CTPACK<sup>7</sup> to pay attention to what kind of knowledge could be required of teachers to teach CT. Other barriers, such as the lack of computers, challenges in rearranging curricula, governmental policies and reluctance to adopt the emerging topic have also been suggested (Lockwood & Mooney, 2018).

Additionally, the topic is so new in compulsory education that the reason why CT is important is not perhaps entirely clearly articulated or well-justified. Teachers may be inclined to teach 'more important' content than programming and CT. Lacking better knowledge, a teacher may ask 'Why must I learn to teach this?' Programming and CT may be seen as optional or extracurricular subjects that are more suited for after-school clubs or elective courses. Bull et al. (2020, p. 13) provided reason for these notions and stated that 'exporting technologies such as Scratch is easier than disseminating the accompanying ideas related to creativity and problem solving that Scratch was designed to support.' The clarification of CT's educational objectives and making it tangible become highly vital.

Additional concerns arise from the perspective of learning CT in the context of programming. Although a recent meta-analysis (Scherer et al., 2020) showed

---

<sup>7</sup> CTPACK is a derivative of the perhaps more well-known TPACK model, which portrays three types of knowledge: technological knowledge, pedagogical knowledge, and content knowledge.

that programming talent is not innate and that programming interventions are generally effective, especially at the primary school level, programming is notoriously difficult for students in many educational settings. Especially motivated students in extracurricular learning activities may have better learning outcomes than other students (Scherer et al., 2020). There are known challenges for educators who attempt to facilitate students' deep learning and 'thinking-doing' for CT (Lye & Koh, 2014). Particularly in the widely adopted block-based programming environments, such as Scratch, in which learners can simply drag and drop pre-assigned programming blocks, there is an ever-looming risk of mechanistic 'doing without thinking' (Ben-Ari, 1998). These issues can be affiliated with the theoretical underpinnings in constructionism, which emphasises discovery-focused learning and hands-on doing (see e.g. Brennan & Resnick, 2012).

Ways to support students' CT learning in programming remains altogether a little studied frontier (Lye & Koh, 2014), and an overall picture of what is involved when considering the pedagogy of CT in Scratch in primary school classrooms remains somewhat unclear. An aspiration to support students' learning in CT can be found in the practice of formative assessment (see Black & Wiliam, 2009), which has been underlined as a key topic of development in CT education (Lye & Koh, 2014).

## 1.5 The focus of this study

Altogether, a rather worrying picture of the current state of including CT and programming in schools presents itself, and it also foreshadows a rather concerning future. In short, there is much novelty enveloping CT; it is a relatively newly defined and rather complicated construct. New kinds of tools (e.g. Scratch) to foster its learning have emerged for teachers, and its inclusion in the context of programming in compulsory education has only begun somewhat recently (Basso et al., 2018; Grover et al., 2017; Heintz et al., 2016). There are few empirical studies focused on examining teaching and learning in this context. Evidence-based knowledge about how students learn the versatile skills and knowledge associated with CT through programming with tools such as Scratch in general classrooms and about where potential pedagogical challenges lie is limited. In particular, it remains largely unclear how students' learning of the multifaceted competence can be supported and what kind of role sociality in learning (e.g. pair programming) can play in the mix.

In response to the growing demands surrounding CT education, this thesis takes the initiative in exploring how the learning of multifaceted CT can be supported in the context of programming with Scratch in primary school classrooms. The main theoretical concept of this thesis is CT, which is perceived to reside at the intersect of computing, programming and problem solving and which can be learnt in the Scratch programming environment meaningfully in primary schools (see also Zhang & Nouri, 2019) (see Figure 2).



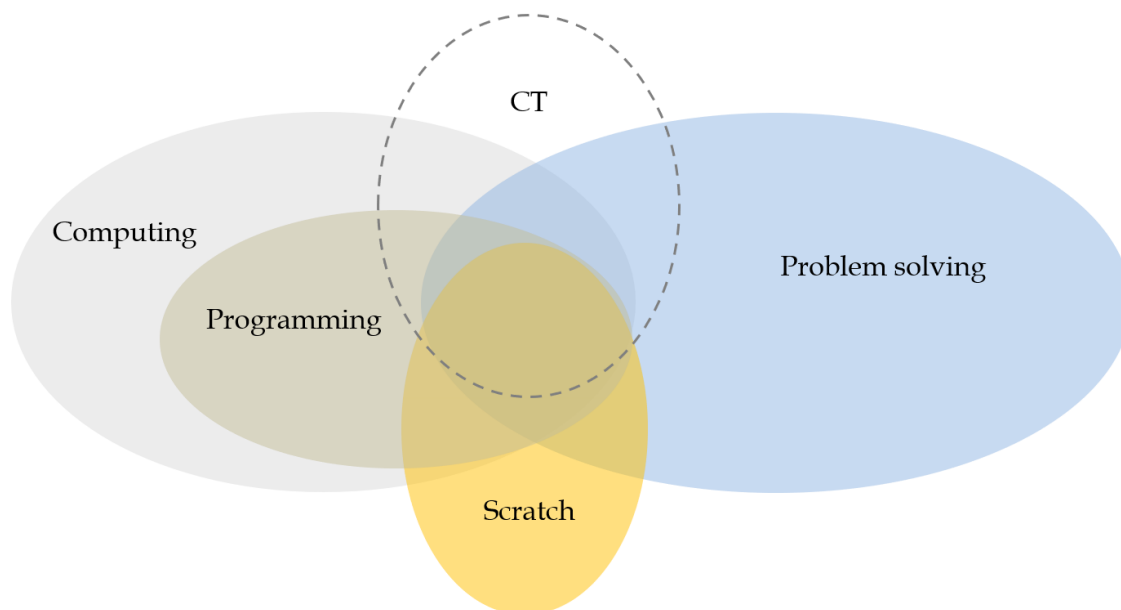


Figure 2. The relationship of key concepts in this thesis (modified from Zhang & Nouri, 2019)

In summary, **computing** is an age-old activity of systematically processing information and can be effectuated by humans and electronic machines alike (Tedre, 2015). **Programming** is a craft-like human activity to design and construct software systems that perform computations in automated programs (i.e. expressions of algorithms that can control the actions of a machine) (Denning & Tedre, 2019; Van Roy & Haridi, 2003). **Scratch** is a web-based programming environment targeted at young learners to creatively design interactive media, such as digital games and animations (Resnick et al., 2009). **CT** is a multifaceted competence that has several proclaimed educational benefits (e.g. providing coding skills, skills for computing across contexts, general problem-solving skills, computational literacy). From there, CT can be understood from different viewpoints that involve potentially different kinds of important skills and knowledge and appropriate pedagogical methods to support learning.

In this study, the focal viewpoint of CT is students' computational **problem-solving** skills that can be examined in the context of Scratch. The main justification is pragmatic: CT is a relatively new topic that has struggled in gaining a solid foothold in primary schools despite it having been accompanied by several well-established and age-appropriate practical tools and methods. There is an urgent need to assist teachers in better adopting CT in practice. CT is introduced in schools typically via programming; therefore, a focus on such teaching and learning activities that encompass problem-solving skills while designing concrete artefacts can provide valuable information regarding how to organise purposeful learning activities in schools.

The overall scholarly and pragmatic aims of the study are to shed light on teaching, learning and assessing CT through programming with Scratch and through this process develop ways to potentially support students' CT learning

with formative assessment in this context. Although formative assessment is not a target of investigation per se, it is an outlying point of reference that particularly motivates the methodological developments herein. On a similar note, the developments regarding assessment are expected to be situated alongside existing research in CT to piece together holistic assessment in CT for further research and development. The concrete actions taken by this thesis to pursue the aims are as follows.

First, this thesis **specifies the educational goals of introducing CT through programming at the primary school level**. Second, by so receiving the requisite theoretical framing for CT and using it, it **evaluates ways to assess students' CT in terms of those goals in Scratch**. These actions stem from the circumstance that no clearly articulated educational objective for general-level CT currently exists in the primary school context. The notions adopted in previous literature concerning what is relevant to teach, learn and assess about CT have thus been inconsistent and rather limited and even programming-centric. This has resulted in a collection of relatively inconsistent and even narrow ways to approach the acquisition of evidence-based knowledge regarding students' potentially relevant skill areas in CT. It has also resulted in ambiguity concerning relevant CT learning contents and activities in Scratch for pedagogical decision making, such as designing programming courses, setting meaningful learning goals and selecting appropriate learning activities to pursue those goals.

Third, this thesis **develops new methods for assessing primary school students' CT richly and holistically in authentic programming situations**. Fourth, following that operational groundwork, it **provides rich empirical insight about students' CT in the context of programming with Scratch**. These actions stem as a consequence from the above-stated disorderly circumstance of the theories and operational measures in CT. There is a lack of rich evidence of how students can learn the various skills and knowledge involved in CT and of what kinds of factors can influence learning in classrooms. This is troublesome primarily for pedagogical decision making, especially in terms of awareness of well-grounded ways to support students' CT learning in classroom contexts. Therefore, the purpose of this empirical work is to explore and describe how students apply CT in practice in this context and learn as many lessons as possible in terms of ways to support their CT learning in Scratch.

The research design, based on the actions taken by this thesis, necessitated both theoretical and empirical efforts. Therefore, a mixed methods design is employed. The efforts are divided into three peer-reviewed scientific journal articles of which the investigator was the first author.

The first article is a literature review. It aims to contextualise CT comprehensively in the Scratch programming environment for teaching and learning in primary school classrooms and to explore the assessment of CT through Scratch in this context.

The second article is an empirical study. It aims to attain rich empirical evidence of primary school students' diverse CT based on Scratch projects they programmed in naturalistic classroom situations. In the process, it also develops a

way to potentially support students' CT in the context of assessing their Scratch projects.

The third article is also an empirical study. Along with Article II, it aims to obtain rich empirical evidence of primary school students' CT. It tackles the issue from the viewpoint of examining students' pair programming processes in Scratch. By taking such an approach, it also paves the way in this rarely employed methodology by developing new analytical methods and discussing how to employ them to support students' CT learning in classrooms.

The above actions taken by this thesis and the roles of each article amidst those actions are summarised in Figure 3.

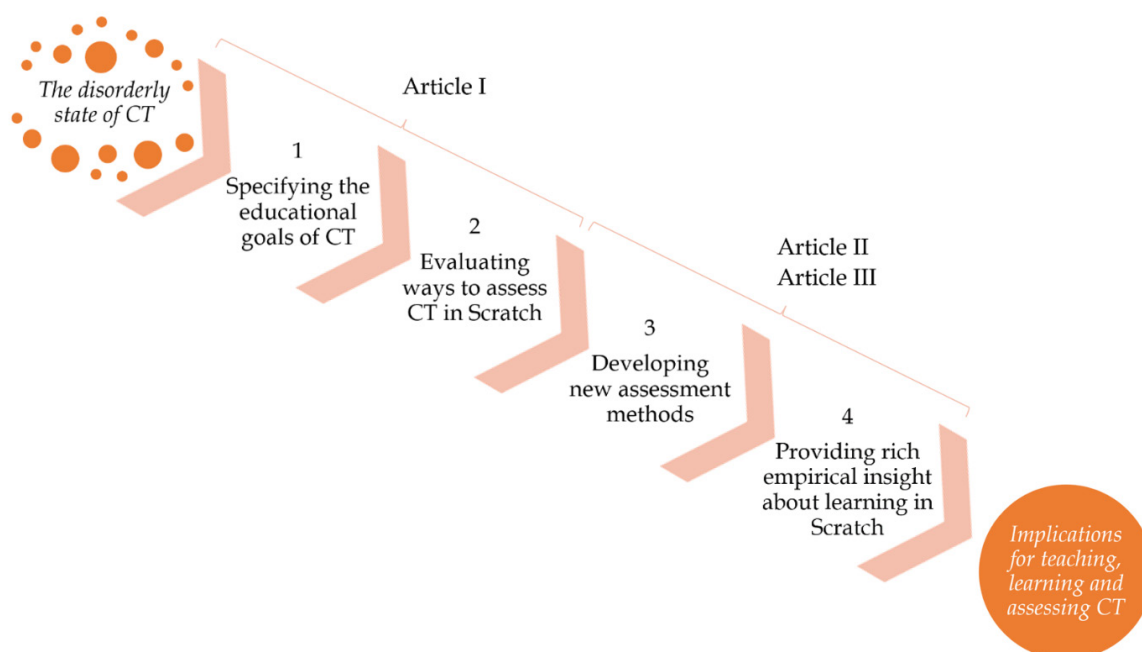


Figure 3. The actions taken in this thesis through the three research articles

## 2 THEORETICAL BACKGROUND

The sections in this chapter present the main theoretical background and literature related to this thesis. Based on previous literature, they describe the conceptual and practical premises related to CT and its role in primary education through programming, which has been often employed in schools as a pathway to teach and learn CT.

The two main sections cover two broad themes that shape the study. The first main section, Computational thinking and programming (section 2.1), clarifies the historical roots of CT, the adopted definition for it, its relationship with programming and ways to concretise it for educational practice. The second main section, Teaching and learning CT in Scratch (section 2.2), overviews ways to implement programming education in schools, especially with the Scratch programming environment. It also explores how CT can be assessed through Scratch projects and shared Scratch programming processes to support students' learning in classroom situations.

### 2.1 Computational thinking and programming

This section focuses on the term 'computational thinking' by examining it specifically in the context of computer programming. To get to the roots of the field and frame the current state of CT, the section begins by situating CT in the historical and contemporary contexts of computing education and ICT education (in subsection 2.1.1). Subsequently, based on previous literature, CT is defined for primary education (in subsection 2.1.2). The relationship between CT and programming—two affiliated conceptual spaces that are important to distinguish—are then clarified (in subsection 2.1.3). Last, the core skill and knowledge areas affiliated with CT are concretised as 'core educational principles' that students can learn by programming (in subsection 2.1.4).

### 2.1.1 A historical overview

Teaching students to think ‘computationally’ when solving problems or in terms of algorithms<sup>8</sup> in different professional fields are not new. According to Grover (2018), several researchers discussed solving problems and thinking using CS decades ago; for example, the problem-solving practices characteristic of CS were discussed in the 1960s, and there was a comparison with ‘CS thinking’ and mathematical thinking in the 1980s.

In fact, CT is a term that Seymour Papert, a renowned computer scientist and educator mainly known for his research on the theory of constructionism and the development of the Logo programming language, first introduced in his book *Mindstorms* (Papert, 1980). Along with contemporary scholars, Papert believed that if a child could teach a computer a real-world phenomenon – such as a formula to solve a mathematical problem, proper syntax to form a sensible sentence or a set of notes in a musical melody – the child would truly understand the phenomenon themselves. Constructionism states that when making ‘things’, learners are implementing (i.e. creating externalisations of) their mental models of how the world works. Children could therefore also learn various subjects by programming matters related to them in an environment that is pedagogically well-suited for their age and the task (Bull et al., 2020).

Perhaps the most well-known example of his ideas, Papert (1980) introduced a two-dimensional virtual turtle that can be programmed with the Logo programming language (see illustration in Figure 4). He argued the meaningfulness of programming the turtle to move in a simulation according to a set of algorithmic rules and to produce various geometrical shapes. By doing so, each individual learner could meaningfully engage in learning that is supported by perfect accuracy and extreme speed at the hands of a digital computer. Blikstein (2020) labelled this as ‘the principle of powerful expressiveness in making’; with the assistance of digital devices, learners’ externalised ideas can come into fruition powerfully, feasibly and very quickly in ‘hands on’ and ‘heads in’ learning activities.

Although Papert’s ideas kindled a wave of contemporary theoretical discussions and empirical research on the learning of computer programming, the term CT was barely used in educational research let alone systematically incorporated in school practice. There was sporadic interest in introducing students to coding in schools in the 1980s and 1990s, particularly with contemporary text-based programming languages that also occasionally adopted tangible technologies such as LEGO/Logo, and in exploratory pedagogical research (e.g. Resnick et al., 1988; Suomala, 1999). According to Kafai and Burke (2013a), the recent re-ignition of coding in schools was fanned by the increased accessibility of easy-to-use computers, especially outside of school. This gave rise to the participatory digital youth culture, an interest-driven culture of making, and connecting with

---

<sup>8</sup> ‘Computational thinking’ is a widely used term in English-speaking contexts, although the term ‘algorithmic thinking’ appears occasionally as a synonym for CT, especially in educational curricula (Bocconi et al., 2018). Moreover, terms such as ‘procedural thinking’ (e.g. Aho, 2011) have been used to in referring to ideas similar to CT.

An algorithm for drawing a square:

1. FORWARD 100
2. RIGHT 90
3. FORWARD 100
4. RIGHT 90
5. FORWARD 100
6. RIGHT 90
7. FORWARD 100

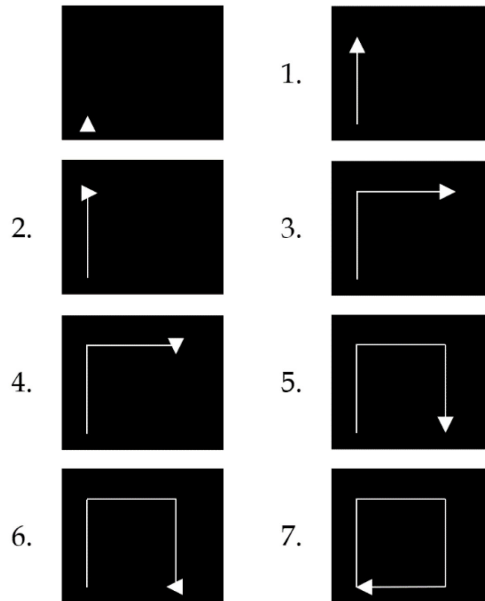


Figure 4. An illustration of the Logo programming environment primarily used in the 1970s

broad networks of other young users, particularly underrepresented ones (e.g. women) (Kafai & Burke 2013a).

Tedre and Denning (2016) view the re-emergence of coding in schools from another perspective. According to an overview of the historicity of CT, the main reason for the increased interest in computing and programming in schools stems from the fact that several scientific domains and sectors of work have begun to digitalise rapidly. Grover (2018) articulate the rise of ‘computational X’, that is, the ‘integration of CT to enable/enrich learning in other disciplines, mainly through the vehicle of programming and automating abstractions and models in other disciplines’. Similarly, Connor et al. (2017) state that CT is largely captured within ‘the understanding of domains that can be modelled by computational mechanisms’. Professionals in disciplines such as computational physics, bioinformatics and digital humanities can perform enormous computations by utilising simulation models and interpreting natural processes as information processes to employ entirely new ways to do science (Denning & Tedre, 2019).

Discourse on how every student should attain adequate ICT-related skills to cope in a future with ubiquitous digital technology dates back decades as well. However, the purpose of CT initiatives, in contrast to mere ICT skill acquisition, is beyond technological applications and hardware. CT is considered to be a broad foundation of thought and a set of skills that is employed autonomously in various authentic problem-solving situations (Bocconi et al., 2017; de Paula et al., 2014). It is more about understanding what *information* is and how it can be identified and modelled as *processes*, in particular with digital computing technology, that establish solutions in real-world situations (Connor et al., 2017). Such ideas partially align with subjects like computer literacy, information literacy, digital literacy, computational literacy and computational fluency, which have had various meanings in history and which can be viewed almost as synonymous

with CT or as closely related to it (see Grover, 2018; Grover & Pea, 2013). For instance, ‘technology comprehension’, a subject that has been incorporated in lower secondary schools in Denmark, combines computing with design and societal reflection, among other topics (Tuhkala et al., 2019). ‘DigCompEdu’, a framework authored by the European Commission (Redecker & Punie, 2017), lays out ‘digital competence’ as an overarching term that can be actualised in several ways, for example, by creating digital content and digital problem solving.

### 2.1.2 Defining computational thinking

Despite more than 10 years of discourse and growing aspirations surrounding CT, it has been plagued by the lack of a stable core description and definition, which has been a hindrance in relation to scientific research and has resulted in ambiguity and uncertainty in practical educational decision making. At present, CT is somewhat ill-structured because there is still no universal agreement about what it constitutes, how it differs from other thinking processes (see more details in subsection 6.3.1) and how to assess it (Tang et al., 2020).

Wing (2006), who popularised CT, originally described it in the following way. CT is ‘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’. Later, she specified that problems and their solutions can be effectively carried out by ‘an information-processing agent’ (Wing, 2011). Since then, CT has been viewed as, for instance:

- a set of skills (e.g. Csizmadia et al., 2015)
- a competence (e.g. Voogt et al., 2015)
- a problem-solving methodology (e.g. Barr & Stephenson, 2011)
- a problem-solving process (e.g. Barth-Cohen et al., 2018; Kalelioglu et al., 2016; Labusch et al., 2019; Standl, 2017) and
- a type of literacy (e.g. Jacob & Warschauer, 2018; Kafai et al., 2019).

Since Wing’s ‘call for action’, there has been much effort to determine what the term could or should mean in practice for professionals working in different fields, for experts in different branches of science and for students studying in comprehensive primary education. Additional attempts to refine or re-define the essence of CT have since appeared. For instance, Aho (2011) specified that an essential component in CT is to represent solutions to problems specifically as ‘computational steps and algorithms’. Shute et al. (2017) also emphasised the algorithmic component, defining CT as ‘the intellectual foundation required to solve problems effectively and efficiently (i.e. algorithmically, with or without the assistance of computers) with solutions that are reusable in different contexts’.

Roughly put, there are two extreme views of what CT provides. First, perhaps in a relatively cautious manner, it provides disciplinary knowledge of CS and the capability to use digital tools in different problem-solving situations (e.g. Denning, 2017). Perhaps it is an exaggeration, but is CT in fact just coding in dis-

guise to prepare students for studies in computational disciplines in higher education? Another view is somewhat more optimistic, which is that CT is virtually equivalent to kinds of generic problem-solving skills that can be applied in a variety of (even non-computational or non-digital) situations (e.g. Wing, 2006). Perhaps this is also an exaggeration, but is CT just a reissue or at most a nuanced depiction of skills that have already been practiced in schools over the ages? To make things more complex, there is the additional question of whether the practical skills in solving problems are more essential for students in terms of CT than the worldview that it can provide. Depending on the emphasis, educators and scholars can choose between focusing more or less on computational problem-solving techniques or understanding, for instance, the social, cultural and ethical matters in the computational world.

Some scholars argue that a broad definition of CT is acceptable, as the attention should focus more on developing teaching, learning and assessing than on defining the 'core' versus the more 'peripheral' qualities in CT (Selby & Woolard, 2014). In educational practice, it may even be sufficient to introduce mere CS to promote hypothetically broader CT (Hu, 2011). Additionally, Voogt et al. (2015) believe that the cognitive processes and practices within CT cannot be defined clearly because they are difficult to implement in practice. Moreover, they emphasise that attempts to define CT very rigidly would elevate a logical definition and dilute the essential idea of CT as a pragmatic approach and make it indistinct from other 21<sup>st</sup> century skills. Yet, other views (e.g. Selby, 2014) underline that an unclear and a disagreed upon definition of CT may hinder CT in gaining a foothold in school curricula and the development of appropriate assessment tools.

Currently, there is relative consensus among scholars and practitioners perhaps in the middle of the extents. What makes CT special is not being mere disciplinary knowledge in CS or going as far as being applicable everywhere but being *the application of computing in different problem-solving contexts*. It is a type of thinking that has roots in the ancient activity of computing and the modern scientific discipline of CS. In other words, CT borrows the concepts, models, ideas and techniques that are part of the discipline of CS. In practice, with the assistance of today's digital computing machines, the core 'habits of mind' involved in CT allow people to produce entirely new kinds of scientific knowledge (e.g. computer-assisted analysis of real-life phenomena), develop new knowledge construction methodologies (e.g. simulations), digitalise societal structures (e.g. automated work) and, most of all, cope within those structures. CT becomes pragmatically meaningful when it manifests as behaviour – computational problem solving. This particular type of problem solving is typically instrumentally supported by ICT tools and devices because automated digital machines are often much better (i.e. faster and more efficient) in carrying out rote computational tasks than humans (Denning & Tedre, 2019; Tedre & Denning, 2016). In fact, 'computational problem solving' has been viewed as an optional label to 'computational thinking' (Grover, 2018).



The praxis of CT is any real-life domain – whether another scientific discipline or a sector of work – that contains a problem that can be solved computationally. Endorsing this view, CT has been visualised as a ‘connecting tissue’ between CS/programming and disciplinary knowledge of the world (Martin, 2018). To utilise CT meaningfully in a problem-solving situation, knowledge of both ends is required – understanding problems and obtaining solutions with the assistance of CS and ICT (Denning & Tedre, 2019; Michaelson, 2015). In conclusion, this study perceives CT as disciplinary knowledge of CS when solving problems with ICT tools in different domains or disciplines (Figure 5).

CT can be acquired by gaining understanding in CS, ICT and problem-solving in addition to knowledge of problems in some real-world domain. Thus, CT is higher-order thinking, which can be exemplified with, for example, Bloom’s taxonomy (Selby, 2015). It does not aim for merely remembering certain CS concepts when using them in operating digital devices. Instead, it is the deliberate application of computational concepts and practices and digital tools in meaningful (and perhaps even new and creative) ways in real-life situations. To include such characteristics tangibly in primary education, this study defines the following educational objective for CT: 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.

Recent studies overviewing curricula in different countries have shown that such pedagogical notions are relevant in schools by way of introducing CS, programming, computing or informatics embedded within different subjects but not for CT specifically (Heintz et al., 2016; Mannila et al., 2014). Students are also expected to acquire particular attitudes or perceptions, such as understanding in computational ethics, through such approaches (Lonka et al., 2018). However, to focus on a manageable theoretical range, this study omits the investigation of students’ more non-cognitive CT-related dispositions and limits its scope to the aspect of computational problem solving.

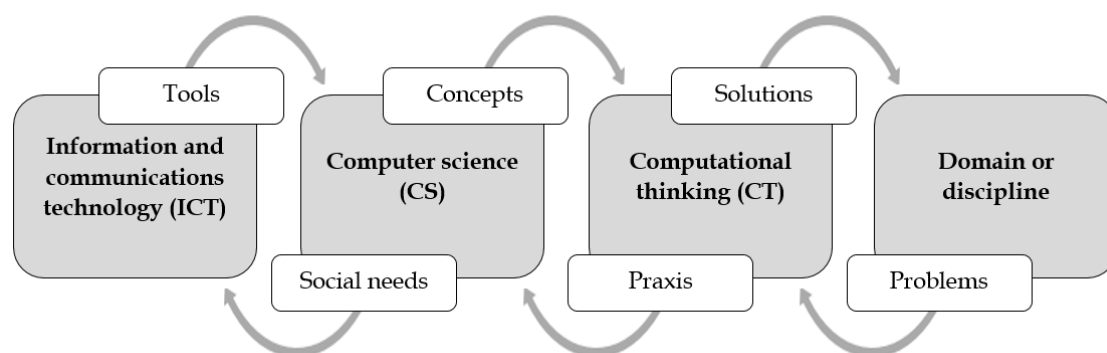


Figure 5. The theory-practice reciprocity of CT and its proximal topics

### 2.1.3 The relationship of CT and programming

To make research and development in CT education tangible, a concrete characterisation of the conceptual ideas and practical skills involved with CT as problem solving is required. For example, Denning and Tedre (2019) qualified key principles that are involved in 'professional CT', that is, the kind of problem solving that CT experts carry out when working professionally in different fields. In a similar fashion, foundations akin to the 'core educational principles' of CT can be formulated to characterise what can be taught and learnt about CT in primary schools.

Several previous works have unpacked CT into different kinds of atomic sub-parts (e.g. key concepts and practices) to specify what skills and knowledge are relevant to it, but those sub-parts have varied across the existing literature (Lye & Koh, 2014; Shute et al., 2017). To that point, the lack of uniform terminology in framing CT's essential sub-parts has also problematised discussions and developing consensual approaches. CT has been said to comprise, for example, key 'facets', 'components' or 'elements' (Shute et al., 2017), 'principles' (Settle & Perkovic, 2010), 'capabilities' (Barr & Stephenson, 2011), 'abilities' (de Araujo et al., 2016) and 'concepts', 'practices' and 'perspectives' (Brennan & Resnick, 2012; Csizmadia et al., 2015; Grover & Pea, 2018).

At its broadest, CT can be viewed as a rich and polymorphous collection of intellect, abilities, attitudes and dispositions. Constraints are in order to investigate it sensibly. This study focuses on investigating primary school students' CT-related problem solving in the context of programming, omitting the 'professional CT' and the more attitudinal, dispositional or perceptive dimensions of it. This rather popular approach adopts a view of CT as involving something that could be referred to as its key concepts (describing 'the what' in what students are dealing with) and key practices (describing 'the how' in how they are performing) (see also Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018; Shute et al., 2017).

Clarifying 'the what' and 'the how' in teaching and learning CT in the context of programming has blurred the vision of many attempting to study and develop CT education. A layer of obscurity is often caused by something that can be described as the concept-spatial ambiguity in CT. The question is: When are we teaching or learning skills that are specifically CT (general, transferable) rather than CS (belonging to that specific scientific discipline), programming (a craft-like human activity) or coding (the act of writing computer language) skills? This notorious problem often seems to have been resolved in previous studies by avoiding being clearly tied to the slippery term that CT is. Instead, studies have often investigated how students learn CS concepts or programming concepts while relying on the assumption that they learn something about CT.

The act of programming can be described broadly as, for example, changing the settings of an alarm clock. Despite this, it is often referred to more rigorously as the craft-like design and construction of software systems that run automated programs in computing machines. When writing programs, programmers use

various data types and logical operations that establish algorithms, which are general-level instructions to solve a specific problem regardless of, for example, which programming language or digital tool is employed to implement said algorithm in a particular situation (Van Roy & Haridi, 2003). It is vital to appreciate that computing and programming are not synonyms; programming is a practical activity whereas computing is a way of understanding computer systems that can be promoted by doing programming (de Paula et al., 2014).

Programming and CT have a rich and intimate relationship. Programming can be examined as an indication of students' CT-related capabilities (e.g. Grover & Pea, 2013), but it is better to examine this relationship in full detail. According to Tang et al. (2020), the key concepts and practices representing CT have been defined in two different conceptual dimensions or hierarchies – context-specific (competences related to a specific practical programming environment) and cross-contextual (competences related to both domain-specific knowledge and more generic problem solving).

The context-specific dimension involves **second-order programming concepts**, which embodies programming and computing-related concepts that have been, for instance, assessed as latent evidence of students' CT skills. Particularly regarding the Scratch programming environment (see subsection 2.2.2), Brennan and Resnick's (2012) famous categorisation of such programming-centric ideas has spawned a myriad of research juxtaposing programming almost directly with CT. The authors proposed that the concepts that CT encompasses and that students should learn to understand and use specifically in Scratch are, for instance, 'loops', 'conditional structures' and 'variables'. In their handbook *Concepts, Techniques, and Models of Computer Programming*, Van Roy and Haridi (2003) portray such concepts as the data types and operations that programmers use to specify systems and design programs that implement those systems. Empirical studies that have examined students' learning of programming with Scratch have referred to similar conceptual ideas as, for example, 'CS subjects' (Zur-Bargury et al., 2013), 'CS concepts' (Meerbaum-Salant et al., 2013), 'programming components' (Seiter & Foreman, 2013), 'language primitives' (Werner et al., 2014) and 'computational concepts' (Moreno-León et al., 2015).

The context-specific dimension rests aside the **first-order cognitive CT** dimension that carries a more cross-contextual interpretation. According to this interpretation, CT skills can be derived from programming concepts, but they are more rooted in cognitive thinking skills that transfer to a variety of computational problem-solving contexts. According to this view, the concepts and practices constituting CT include interdisciplinary capabilities or competence areas instead of skills and knowledge specific to mere programming (or any other practical problem-solving context) (Tang et al., 2020). This view aligns with the notion that the kind of CT that students should develop is aimed to be applicable in different areas of life and work (e.g. Wing, 2006). As Patton et al. (2019) put it, the decontextualisation of computing from real-world contexts and applications can threaten to decrease students' understanding of the relevance of its utilisation in their future.

Framing CT's core constituents in a narrowly scoped practical context may prove to be limiting in terms of its potential transferability across different computational contexts. Therefore, this thesis' view is that CT's key concepts and practices are not by default conceptually aligned to a particular practical context, as in the context-specific dimension, but that they can potentially be applied in various contexts, as in the cross-contextual interpretation. However, as programming is a meaningful pathway to develop in CT, it can comprise context-specific elements that can influence the development of cross-contextual CT concepts and practices. Therefore, the relationship between programming and CT is understood in this thesis as a conjoined one. In programming, *students' programming activities and the programming contents they encounter can foster certain areas within CT*. In turn, *students' CT can be assessed by observing their programming work* because the computer programs that students design to solve particular problems can indicate the conceptual and practical encounters they have had and suggest their capabilities and progress in CT (Brennan & Resnick, 2012; Denning, 2017; Grover & Pea, 2013, 2018; Kafai & Burke, 2013b; Seiter & Foreman, 2013). This framing is further concretised in Scratch in subsection 2.2.5.

#### 2.1.4 CT's core educational principles

It is necessary to establish what constitutes the first- and second-order dimensions in the relationship between CT and programming. Prior literature has framed the more cross-contextual key concepts and practices in CT in different ways. Building from the *Great Principles of Computing*, Settle and Perkovic (2010) presented a framework for CT across the curriculum in undergraduate education. They proclaimed that CT involves such principles and keywords as 'communication', 'automation' and 'design' that organise the pivotal instances of CT that can translate to contexts outside CS.

In 2009 (and later in 2018), the International Society for Technology in Education (ISTE) and the Computer Science Teachers Association collaborated to devise an operational definition for CT in K-12 classrooms. Among their core ideas concerning what CT is were concepts and methodologies that students use when solving problems with a computer, such as 'data collection', 'abstraction' and 'algorithms and procedures'. Such concepts were embedded in activities across multiple disciplines (Barr & Stephenson, 2011; ISTE, 2018).

In the aftermath of computing being introduced in British schools in 2014, Csizmadia et al. (2015) developed a conceptual framework for CT that involved concepts such as 'logical reasoning', 'thinking algorithmically' and 'decomposition'. They intended to help teachers to incorporate CT stemming from prior efforts in computing and CS education in the *Computing at School* projects that would also support learning and thinking in other curricular areas. Similarly, Angeli et al. (2016, p. 50-51) designed a K-6 CT curriculum comprising CT skills and implications for teacher knowledge.

In a subsequent attempt to demystify CT to support the educational reforms spurred by the topic globally, Shute et al. (2017) reviewed prior literature and synthesised such CT facets as 'decomposition', 'iteration' and 'generalisation' as

ways of thinking and acting that students can display by developing and using certain skills. Similarly, Hsu et al. (2018) reviewed prior literature and discussed how CT could be taught and learnt in K-12. Furthermore, based on recent widespread discourse on CT, Grover and Pea (2018) attempted to bring clarity regarding what CT is and what it is not by providing their own categorisation of such CT concepts and practices as ‘logic and logical thinking’, ‘evaluation’ and ‘creating computational artefacts’.

Altogether, the previous works provide kindred yet not entirely convergent categorisations of the key concepts and practices that compose the theoretical space of cross-contextual CT (and the terminology by which they are referred to) (see Figure 6). It is worth noting, though, that CT is an elusive term that continues finding an established form, 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 general-level CT. In turn, generous views may overlap with other competence areas. By framing CT based on several previous works, this study strives to adopt a relatively generous rather than a narrower view to expand our understanding of the potentially meaningful borders of CT. This approach can be feasibly limited, as needed.

Researchers have carried out detailed investigations into how students can learn the various key concepts and practices of CT, such as abstraction (e.g. Liebe & Camp, 2019; Statter & Armoni, 2020; Waite et al., 2018), decomposition (e.g. Rich et al., 2018), data variables (e.g. Rich et al., 2020) and algorithms (e.g. Dwyer et al., 2014). In fact, debugging was examined several decades ago in programming education (e.g. Klahr & Carver, 1988; Pea et al., 1987; Vessey, 1985) and with contemporary environments (e.g. Liu et al., 2017). CT has also been viewed specifically through the distinct disciplines or practices of ‘modelling’ (Sengupta et al., 2018) and ‘design’ (Oleson et al., 2020). Then again, pattern recognition and generalisation can be viewed as immensely vast concepts because patterns exist everywhere, and recognising and generalising based on them are cognitive processes that likely transcend all contexts of human experience and cognition.

Altogether, the key concepts and practices are broad and qualitatively different, testifying to the catalysts for the infamous complexity and an unclear consensus regarding CT. From there, it is important to consider what the concepts and practices mean, especially in programming, and how they can be viewed as



Figure 6. Key concepts and practices that prior literature associates with CT

a holistic but dismantlable collection of ideas, models and techniques that can be taught and learnt conveniently and studied systematically. Few prior works have systematically explored the relationships between CT and programming. Those that have include Selby (2015), who theorised a general model that parallels Bloom's taxonomy with a few select CT concepts (e.g. decomposition, abstraction) and programming skills (e.g. code constructs, an algorithm). The model presents a hierarchy of cognitive complexity in CT and programming and thus teaching order that, in short, begins by students learning to comprehend basic code constructs and ending with them having skills to evaluate algorithms. Kong (2016) proposed a framework according to which specific areas in CT are delivered through a number of learning projects, which are used to review students' extent of understanding. Zhang and Nouri (2019) also reviewed on a relatively general level how different programming elements relate to specific CT concepts and devised a progression for them for certain age groups.

Although these works are on point, they have remained rather programming-centric or too generic to account for several important conceptual and practical areas that can be regarded as important in CT education in practice. Various programming contents that students manipulate and programming activities that they carry out can foster the skills and knowledge involved inclusively in CT in different ways. Fortunately, the research categorising CT's key concepts and practices has characterised the skills and knowledge included in CT that students can gain while programming in manifold ways. To concretise said skills and knowledge, this study summarises them uniquely as **CT's core educational principles** (CEPs) – fundamental computational facts, conceptual ideas and techniques; atomic elements of CT to enable the systematic contextualisation of CT in specific programming environments. The CEPs derived from previous literature are as follows:

- **Abstraction.** A range of digital devices can be computers that run programs (Csizmadia et al., 2015; Grover & Pea, 2018). Programming languages, algorithms and data are abstractions of real-world phenomena (Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018). Solving complex problems becomes easier by reducing unnecessary detail and by focusing on parts that matter (for example, by using data structures and appropriate notation) (Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018).
- **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 (Angeli et al., 2016; Barr & Stephenson, 2011; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018; Settle & Perkovic, 2010; Shute et al., 2017). Sequencing, selection and repetition are the basic building blocks of algorithms (Angeli et al., 2016; Barr & Stephenson, 2011; Csizmadia et al., 2015; Grover & Pea, 2018). Recursive solutions solve simpler versions of the same problem (Barr & Stephenson, 2011; Csizmadia et al., 2015; Grover & Pea, 2018).

- **Automation.** Automated computation can solve problems (Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018). Programmers design programs with computer code for computers to execute (Csizmadia et al., 2015; Grover & Pea, 2018; Settle & Perkovic, 2010). Computers can use a range of input and output devices (Csizmadia et al., 2015).
- **Collaboration.** Programmers divide tasks and alternate in roles (Grover & Pea, 2018). Programmers build on one another's projects (Angeli et al., 2016; Grover & Pea, 2018). Programmers distribute solutions to others (Grover & Pea, 2018).
- **Coordination and Parallelism.** Computers can execute divided sets of instructions in parallel (Barr & Stephenson, 2011; Csizmadia et al., 2015; Hsu et al., 2018; Shute et al., 2017). The timing of computation in participating processes requires control (Settle & Perkovic, 2010).
- **Creativity.** Programmers employ alternate approaches to solving problems and 'out-of-the-box thinking'. Creating projects is a form of creative expression (Grover & Pea, 2018).
- **Data.** Programmers find and collect data from various sources and multilayered datasets that are related to each other (Barr & Stephenson, 2011; Hsu et al., 2018; Shute et al., 2017). Programs work with various data types (e.g. text, numbers) (Barr & Stephenson, 2011; Csizmadia et al., 2015; Hsu et al., 2018). Programs store, move and perform calculations on data (Angeli et al., 2016; Barr & Stephenson, 2011; Csizmadia et al., 2015; Settle & Perkovic, 2010). Programs store data in various data structures (e.g. variable, table, list, graph) (Angeli et al., 2016; Barr & Stephenson, 2011; Csizmadia et al., 2015).
- **Efficiency.** Algorithms have no redundant or unnecessary steps (Csizmadia et al., 2015; Shute et al., 2017). Designed solutions are easy for people to use (Csizmadia et al., 2015). Designed solutions work effectively and promote positive user experience. Designed solutions function correctly under all circumstances (Csizmadia et al., 2015; Grover & Pea, 2018).
- **Iteration.** Programmers refine solutions through design, testing and debugging until the ideal result is achieved (Grover & Pea, 2018; Shute et al., 2017).
- **Logic.** Programmers analyse situations and check facts to make and verify predictions, make decisions and reach conclusions (Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018). Formulated instructions comprise conditional logic, Boolean logic, arithmetic operations and other logical frameworks (Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018).
- **Modelling and design.** Programmers design human-readable representations and models of an algorithmic design that can later be programmed (Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018; Shute et al., 2017). Programmers organise the structure, appearance and functionality of a system well (Csizmadia et al., 2015; Settle & Perkovic,

2010). Visual models, simulations and animations represent how a system operates (Angeli et al., 2016; Barr & Stephenson, 2011; Csizmadia et al., 2015; Hsu et al., 2018).

- **Patterns and Generalisation.** Data and information structures comprise repeating patterns based on similarities and differences in them (Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018; Shute et al., 2017). Repeating patterns form general-level solutions that apply to a class of similar problems (Barr & Stephenson, 2011; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018; Shute et al., 2017). General-level ideas and solutions solve problems in new situations and domains (Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018; Shute et al., 2017).
- **Problem decomposition.** Large problems and artefacts decompose into smaller and simpler parts that can be solved separately (Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018; Shute et al., 2017). Large systems are composed of smaller meaningful parts (Angeli et al., 2016; Grover & Pea, 2018). Programs comprise objects, the main program and functions (Barr & Stephenson, 2011).
- **Testing and debugging.** Programmers evaluate and verify solutions for appropriateness according to their desired result, goal or set criteria. Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018). Programmers evaluate solutions for functional accuracy and detect flaws using methods involving the observation of artefacts in use and comparing similar artefacts (Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018; Shute et al., 2017). Programmers trace code, design and run test plans and test cases and apply heuristics to isolate errors and fix them (Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018; Hsu et al., 2018; Shute et al., 2017). Programmers make fair and honest judgements in complex situations that are not free of values and constraints (Csizmadia et al., 2015).

## 2.2 Teaching and learning CT in Scratch

This section begins by overviewing the growing number of programming environments intended for introducing CT and programming in schools (in subsection 2.2.1). In particular, the section explains why creative computing in Scratch was selected as the focal programming context to examine students' CT in this study. The capabilities of Scratch as a programming tool and its background pedagogical ideology are described (in subsection 2.2.2).

Subsequently, previous research on the pedagogy of programming is scoped (in subsection 2.2.3). In particular, the section highlights the disordered state of empirical knowledge regarding students' learning of multifaceted CT in



Scratch in classrooms. Consequently, the need for a more systematic understanding of what is involved when examining students' learning and ways to support it is suggested. In that regard, the section then elaborates the pedagogical interest labelled 'assessment for learning' (in subsection 2.2.4) to position the developments in this study in an organised and potentially functional manner. These developments include two assessment viewpoints selected for this study – aiming to support students' learning by assessing the CT-fostering programming contents in their Scratch projects ('the what') and by assessing CT-fostering programming activities in their collaborative programming processes in Scratch ('the how') (in subsection 2.2.5).

### 2.2.1 Scratch amid contemporary programming environments

Programming education often adheres to project-based construction of external artefacts in environments emphasising personalised learning, active searching and discovery (Brennan & Resnick, 2012; Papert, 1980; Resnick et al., 2009). Multiple kinds of contemporary environments and contexts have been developed for students to learn programming and CT in schools and beyond (see sample environments in Figure 7). They include robotics (e.g. Barth-Cohen et al., 2018; Chalmers, 2018), physical and unplugged tools (e.g. Brackmann et al., 2019), game development (e.g. Denner et al., 2012), app design (e.g. Papadakis et al., 2017) and 'digital fabrication' (e.g. Iwata et al., 2020; Suero Montero, 2018). The former is also referred to as the 'maker movement', which emphasises hands-on doing (e.g. tinkering) and problem solving in technology-rich environments (e.g. Korhonen & Lavonen, 2015; Sormunen et al., 2019).

Reinforcing Wing's (2006) original portrayal of CT highlighting that both humans and digital devices can carry out computational tasks, empirical research has found that conceptual learning in CT and programming can occur in non-programming situations aside from mere programming (Grover et al., 2019). Therefore, using programming or even computers is not always even mandatory to develop CT. For instance, Twigg et al. (2019) presented a creative story-based pedagogy approach to introducing key CT concepts to small children through children's literature. Haroldson and Ballard (2020) gathered several CS-related children's picture books and graphic novels and found that they largely included notes of computational practices, such as developing and using abstractions and creating computational artefacts. Bers et al. (2019) successfully introduced computational concepts to 3-year-old children through engaging tangibly programmable robots. Game-based learning, that is, learning by playing video games, has been relatively long recognised as a pathway to learn CT (e.g. Ch'ng et al., 2019; Gibson & Bell, 2013; Kazimoglu et al., 2012). Additionally, educators have employed various approaches, including kinaesthetic ones (e.g. the 'unplugged' movement, such as play acting<sup>9</sup>) to promote authentic and exciting learning

---

<sup>9</sup> 'Harold the Robot' (<https://classic.csunplugged.org/harold-the-robot-2/>) (and others akin to it) can perhaps be considered one of the most popular play-like activities that have been used to begin computational learning with young students.

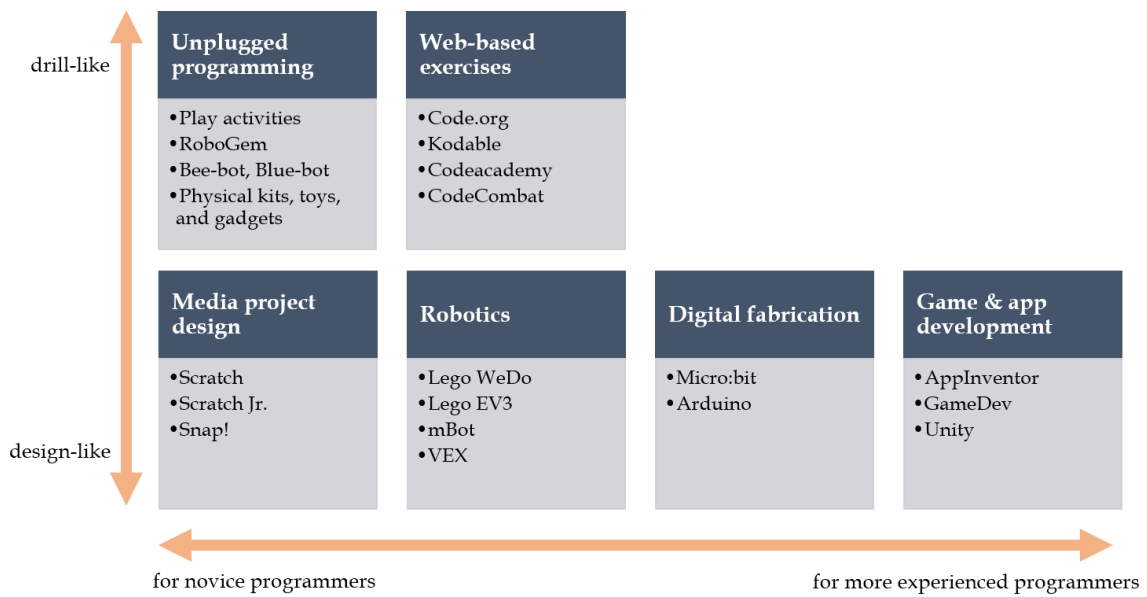


Figure 7. Some contemporary programming environments utilised in schools

experiences, facilitate hands-on making and support even early childhood education students and students with special needs (Del Olmo-Muñoz et al., 2020; Garneli et al., 2015).

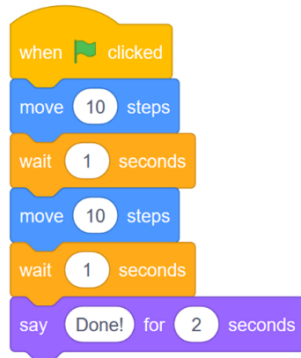
Programming in the aforementioned different ways could be seen to occur in a type of 'microworld' that can have unique characteristics in terms of the programming language used, the thematic setting, the design limitations and the computational concepts available for use (Pelánek & Effenberger, 2020). Different kinds of programming environments may thus contribute more or less strongly to learning different areas in CT (Park & Shin, 2019). In particular, the relationship of 'non-programming' (e.g. unplugged exercises) and CT is not entirely straightforward. On one hand, Grover et al. (2019) found that conceptual learning in non-programming situations prior to actual programming significantly improved learning gains among primary school students. On the other hand, utilising the ever-developing power of modern computing machines and designing algorithms that control them in order to address human concerns are the naturalistic and rational goals of CT (Denning, 2017; Denning & Tedre, 2019). Somewhat more practically, Huang and Looi (2020) contemplated that CT is separated from programming in unplugged approaches, but this separation can influence the definitions and instruction in CT. For instance, how can CT, a competence encompassing computational problem solving with digital computers, be assessed without using programmatic representations? On an entirely other note, Bull et al. (2020) pointed out that a small percentage of technology purchased for schools is altogether even used. Introducing CT with specially bought devices can therefore be fiscally irresponsible. Furthermore, considering that the time available for CT instruction in schools is usually constrained, the use of digital computers as deliverers of automated feedback raises the stakes (Bull et al., 2020).

Although non-programming approaches have increased in popularity in CT education, digital computer programming has perhaps been a more established route to facilitate activities that foster students' acquisition of CT at several educational levels (Grover & Pea, 2013; Lu & Fletcher, 2009; Voogt et al., 2015). Programming a digital computer is activating, artefact-oriented work the goal of which is to design technological systems and programs that can be used (Grover & Pea, 2013; Van Roy & Haridi, 2003). Programming environments enabling such work encompass designing computer code with text-based languages, that is, with text, numbers and symbols (Michaelson, 2015). Yet, graphical programming languages have long been utilised as pathways to begin learning programming through positive experiences with younger students and novice learners (Navarro-Prieto & Cañas, 2001; Resnick et al., 2009; Taylor et al., 1986), even in higher education (e.g. Malan & Leitner, 2007). Graphical programming languages are at a higher level of abstraction, and they include much more pictorial information in contrast to text-based languages. Typically, programs are designed with the provided visual code blocks that are combined to accomplish specific computational operations (see comparison in Figure 8). Key pedagogical premises in graphical languages are that they make abstract computational ideas more concrete, provide a limited set of computational operations, eliminate the possibility of syntactic mistakes (i.e. typing errors), promote a feeling of tinkerability and direct more focus to higher-order concepts instead of technical details<sup>10</sup> (Bull et al., 2020; Maloney et al., 2010). Graphical programming languages have even been found to minimise students' misconceptions of particular computational models (Mladenović et al., 2018). To that point, the Finnish national primary school core curriculum explicitly states that students aged 9–12 should practice programming in a graphical programming environment (Opetushallitus, 2014, p. 235).

Environments for learning CT (programming-focused and non-programmatic alike) differ in terms of, for example, whether they are either stand-alone or more akin to programming platforms and whether they come with integrated learning support (e.g. tasks, feedback). Although particular environments can be viewed as contexts to merely become acquainted with elementary-level CT and programming, CT has the expectation to be a competence that transfers to other computational problem-solving contexts in different scientific disciplines (or school subjects) and real-life contexts. Therefore, it should not ultimately operate only as an isolated target for learning but as a way to shape learning methods and learning processes by combining subjects in non-traditional and engaging ways, as emphasised in contemporary views of authentic, multidisciplinary learning (Lonka et al., 2018). In fact, several studies have explored in theory and

---

<sup>10</sup> Currently, the effect of graphical programming environments on cognitive learning outcomes versus text-based ones is not entirely clear (Xu et al., 2019). However, a recent meta-analysis (Scherer et al., 2020) confirmed that although visual languages may convey distractive elements, visualization in programming languages has a moderate effect on learning, likely by means of reducing cognitive load and aiding the creation of mental models through visual representations.

**Graphical (or block-based) programming****Text-based programming**

```
input.onButtonPressed(Button.A, function () {
  music.playTone(262,
  music.beat(BeatFraction.Whole))
  basic.pause(1000)
  music.playTone(330,
  music.beat(BeatFraction.Whole))
  basic.pause(1000)
  basic.showString("Done!")
})
```

Figure 8. Similar programs in two programming languages (left: Scratch, right: JavaScript in Micro:bit)

practice how activities encompassing different elements in CT can be meaningfully integrated across entire curricula (e.g. Dong et al., 2019; Isbell et al., 2009; Israel et al., 2015; Perković et al., 2010; Settle & Perković, 2010). CT has also been more closely examined within specific subjects, such as:

- language studies (e.g. Weng, 2018; Whyte et al., 2019)
- scientific inquiry (e.g. Basu et al., 2014; Hutchins et al., 2018; Luo et al., 2020; Sengupta et al., 2013; Swanson et al., 2019; Weintrop et al., 2015; Yadav et al., 2018)
- crafts (e.g. Kafai et al., 2019; Lui et al., 2018) and
- math (e.g. Kahn et al., 2011; Kong, 2019; Promraksa et al., 2014; Tan et al., 2019).

In practice, CT has often been introduced in STEM (Hutchins et al., 2018; Kafai et al., 2019) or science, technology, engineering, the (liberal) arts and mathematics (STEAM) (e.g. Pears et al., 2019) contexts through topics such as robotics, web development, circuit boards and product design or software engineering (Lockwood & Mooney, 2018). For instance, Nijenhuis-Voogt et al. (2020) interviewed teachers and found that they had taught algorithms in various contexts, including professional or scientific (e.g. solving professional problems), everyday life (digital creativity) and societal contexts (e.g. through real-world analogies). Problem-based learning, project-based learning, collaborative learning and game-based learning have shown to be the most common approaches adapted in previous studies (Hsu et al., 2018). Notably, however, although the incorporation of CT through contexts such as design and robotics has increased students' interest and engagement (e.g. Luo et al., 2020), it has yet to show encouraging empirical results in terms of effectively learning subject matter (Tang et al., 2020). There is a known need for longitudinal and controlled studies identifying the effectiveness of pedagogical approaches in CT education (Szabo et al., 2019).

Particularly in environments emphasising designing and making rather than drilling and solving pre-set exercises, students' learning in programming can be theorised by constructionism (see more in subsection 2.2.2), which essen-

tially involves the active design of personally meaningful concrete artefacts. Notions emphasised in 21<sup>st</sup> century education have emphasised the importance of promoting such learner-centred, collaborative and multidisciplinary project-based learning experiences for students (Lonka et al., 2018; Opetushallitus, 2014). In terms of programming, Kafai and Burke (2013a) raise three points of shift – shifting from computer code in itself to meaningful applications and their making, shifting from mere programming environments to communities of making and shifting from creating ‘from scratch’ to creating via ‘remix’. Blikstein (2020) discusses ‘the emancipatory principle in making’: an opportunity to reconnect learning to the real world to allow students to express their ideas and participate in the world of computing on their own terms. CT learning through programmatic design can indeed be driven by students’ interest areas and enjoyment in making (Brennan & Resnick 2012). Therefore, it is justifiable for students to participate in multidisciplinary project-based learning activities that can have real-world connections. One especially popular, versatile and accessible (e.g. free to use) environment facilitating such activities is Scratch (Hsu et al., 2018).

### 2.2.2 Creative computing with Scratch

CT is examined in this thesis in the context of programming with Scratch, a web-based programming environment that originated<sup>11</sup> at the Massachusetts Institute of Technology Media Lab. In Scratch, students can combine various thematically coloured graphical blocks to establish sets of algorithmic instructions (‘scripts’) that can, when semantically valid, produce creative behaviours (e.g. animations) for digital characters (‘sprites’, such as a penguin or a fox) on the computer screen (Figure 9) (Maloney et al., 2010; Resnick et al., 2009). There is also a stripped version of Scratch called ScratchJr, which has made graphical programming and the basics of CT available and meaningful to even very young students, such as preschool aged children (e.g. Papadakis et al., 2016). ScratchJr has been utilised in schools especially in early childhood education for acquainting young children with block-based digital media design and assisting the transition from more playful coding (e.g. unplugged approaches) to environments utilising coding languages, such as graphical programming with Scratch (Bers et al., 2019).

The developers of Scratch designed the environment to improve the technological fluency of children attending after-school centres in economically disadvantaged communities in the United States (Resnick et al., 2003). Since its launch, Scratch has spread to more than 150 countries, has been translated into more than 40 languages and has been used by more than 26 million registered users who have shared more than 30 million programming projects, according to

---

<sup>11</sup> The first version of Scratch was released in 2007, and a second version was publicly released in 2013. The current version, Scratch 3.0, was released in 2019, when empirical data for this study was collected. However, the results attained in this study with the 2.0 version of Scratch are still valid for the newest version, as no major functional changes were made to the 3.0 version. The current version is free to use on its website: <https://scratch.mit.edu/>

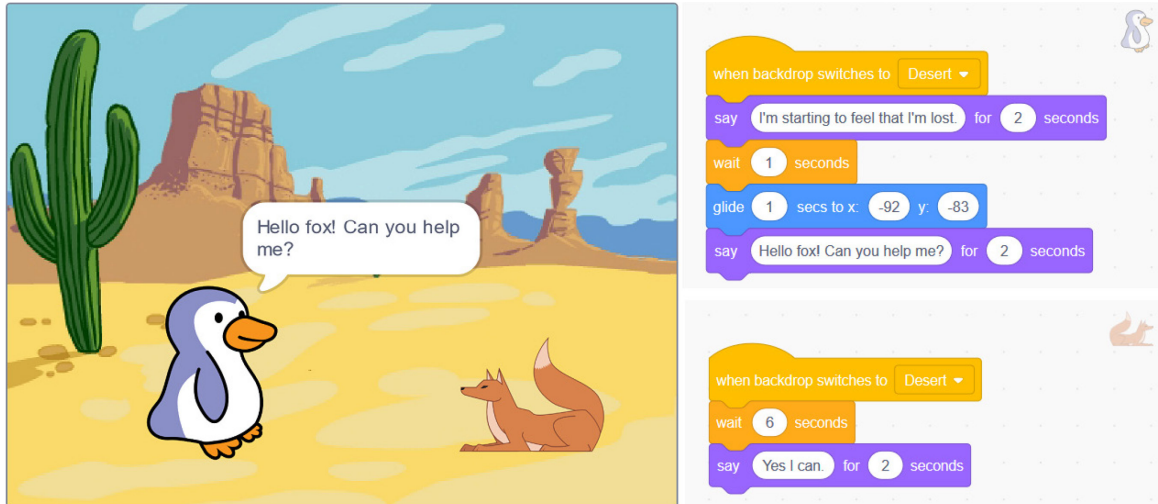


Figure 9. Sample screenshots from Scratch

current statistics on its website<sup>12</sup>. In addition, it is by far the most often employed programming language in scientific research (Hsu et al., 2018).

Prior to the development of Scratch, several educators saw that there were particular problems with the available programming languages. Children struggled with the programming language syntax and the fact that the affordances that the tools offered were mainly detached from children's interests (Bull et al., 2020). Consequently, the developers of Scratch aimed to enable the design of diverse projects, such as games, stories, animations, simulations and music/art performances by promoting creativity and personalisation. The block-based coding is supposed to feel 'tinkerable' to promote a feel of playing, building and evolving plans, goals, structures and stories organically. Scratch also aims to be more social; it is a public web-based social platform where users share their projects publicly and view, use, comment and remix others' projects (Resnick et al., 2009). Its informality has been promoted by supporting computational creation in and out of school (Brennan, 2013). Besides the intended benefits, the actual effect of using Scratch in teaching and learning programming was recently discovered to be significantly greater than other languages<sup>13</sup> (Scherer et al., 2020).

Programming with Scratch is rooted in the constructionist approach to learning (Brennan & Resnick, 2012). According to Ackermann (2001), Papert based the theory of constructionism as a kind of 'art of learning' on Piaget's epistemological theory of constructivism. In particular, Papert underlined that students learn and build new knowledge on top of their prior knowledge by actively participating in discovery and creating tangible artefacts in project-based settings. According to Papert, the relevance of new knowledge and personal interest

<sup>12</sup> <https://scratch.mit.edu/statistics/>

<sup>13</sup> Although visibility was confirmed to have an effect on learning, an even larger effect was found in physicality (e.g. robotics). This finding suggested that the immediate feedback gained from observing the behavior of programmed physical tools and motivation for engaging in coding (Scherer et al., 2020).

should be at the heart of the learning process to make learning as effective as possible (see Papert, 1980; Papert & Harel, 1991).

In light of Papert's constructionism, Scratch provides a tool for students to act as creative imaginers and designers and active manipulators of data and programmatic operations that establish meaningful computational systems for a creative purpose (Brennan & Resnick, 2012). This is also known as 'creative computing', an approach that emphasises self-expression, imagination and personal interest areas (Brennan et al., 2014). In practice, students have used Scratch with more drill-like, scenario-like or open-ended problem/project-based approaches to design multidisciplinary, interactive media projects that are thematically connected to various curricular areas (Garneli et al., 2015; Hameed et al., 2018; Robles et al., 2018; Saéz-López et al., 2016). Among concrete examples are designing creative projects to learn about numerical sequences, environmental science and storytelling (Moreno-León & Robles, 2016). Such projects have even been discovered to have a positive effect on students' motivation to study subject content (e.g. Weng, 2018).

### 2.2.3 Pedagogical underpinnings in graphical programming

Previous research concerning the pedagogy of programming spans multiple educational levels and programming contexts (e.g. kinds and purposes of learning tasks), contributing to the versatility of key pedagogical underpinnings concerning, for instance, learning objectives, instructional strategies and guidance methods. Although only relatively few studies on teaching and learning specifically cross-contextual CT in Scratch or other graphical programming environments have been conducted in recent years, relevant pedagogical underpinnings can be adopted from more context-independent educational research in programming and empirical studies of other programming environments. In fact, the theoretical roots of programming – graphical or otherwise – can be traced to the foundations of cognitive psychology (e.g. Pea & Kurland, 1984) and constructionism (Papert, 1980).

Learning CT through programming can be viewed through the process of 'conceptual change', that is, having some kind of previous knowledge and experiencing an assimilation (conceptual capture) or accommodation process (conceptual exchange) (Duit & Treagust, 2003). When considering the pedagogy in CT in the context of programming, it is important to note that programming involves various tasks, some of which can be considered rather complex. In addition to the mere design of code, programming includes planning and organising code, monitoring the problem-solving process and success, self-reflection and troubleshooting (Scherer et al., 2020). Furthermore, problem solving with CT in programming requires interpreting and navigating multiple representations of information across, for instance, task instructions, programming interfaces and output (Barth-Cohen et al., 2018). It requires operating on multiple levels of abstraction and thinking in terms of complex systems with several co-operating logical components (Csizmadia et al., 2015). These facts inevitably establish a complex system of areas in which conceptual change can occur in programming.

Although graphical programming environments, such as Scratch, may be more effective for young novices when starting to learn programming (Navarro-Prieto & Cañas, 2001; Resnick et al., 2009; Taylor et al., 1986), learners build new knowledge beyond the pictorial information only when the icons representing the computational objects come to establish new mental models (Ben-Ari, 1998). In fact, there is nascent evidence that CT interventions may not always have the desired effect on students' conceptual learning (e.g. Bers et al., 2014; Luo et al., 2020). More precisely, misconceptions and difficulties in students' learning of certain computational notions have been empirically discovered (e.g. Burke, 2012; Franklin et al., 2013; Maloney et al., 2008; Meerbaum-Salant et al., 2013; Seiter, 2015; Seiter & Foreman, 2013; Swidan et al., 2018).

Although approaches such as blended learning (i.e. using both online and traditional materials and methods) and game-based learning (i.e. playing games to learn) were recently found to have slightly larger effect sizes in learning, there is no currently known, explicitly best instructional approach to facilitate the conceptual learning of computer programming (or, rather, there are several effective ways to teach and learn it) (Scherer et al., 2020). Previous research on Scratch specifically with young learners has revealed a positive effect of design-based learning, encouraging the facilitation of more constructionist-like learner-centred doing rather than direct instruction of knowledge (e.g. Jun et al., 2017).

The built-in emphasis on constructionist learning in Scratch largely encourages facilitating learning by discovery in creative, open-ended scenarios with little deliberate instruction. Such settings have several voiced benefits, such as increasing motivation, adapting to versatile learning strategies and increasing the relevance of learning by introducing new knowledge when it is concretely needed (Brennan & Resnick, 2012). Previous educational research has emphasised learners' individual cognition, but the social dimension in learning has recently been emphasised as enhancing learning as well (Roschelle & Teasley, 1995). A salient learning theoretical underpinning is collaborative learning through programming in pairs (or small groups). Students programming artefacts together have a joint goal, and they also share meanings and build knowledge together by, for instance, asking questions, explaining and comparing viewpoints (Arisholm et al., 2007; Preston, 2005; Wei et al., 2021). Highly collaborative and engaging programming activities have been found to significantly moderate children's attitudes towards learning programming (Sharma et al., 2019). In turn, highly structured rather than tinkering and fantasy-focused learning tasks have been found to have a negative impact on students' interest in programming (Dohn, 2019). These issues are important to consider because interest in programming indicates higher programming self-efficacy, and better attitudes toward collaboration indicate higher creative self-efficacy (Kong et al., 2018).

However, it is important to take note of the criticism for foundational ideas in more discovery-focused learning (see e.g. Mayer, 2004; Kirschner et al., 2006). Historically, disciplines have been taught as a body of knowledge; however, there has been occasional advocacy for pedagogical approaches that encourage



relatively unguided practical or project-oriented work, thus rejecting the structured presentation and explanation of disciplinary content. Research has not shown support for instruction using minimal guidance, whereas it nearly exclusively supports direct instructional guidance, especially with novice to intermediate learners (Kirschner et al., 2006). According to Mayer (2004), 'minimal to no guidance' does not, however, problematise the active learner but the seemingly inactive instructor. Lack of guidance in pure discovery leads to unconstructive or even misled construction of new knowledge. Based on decades of research on problem solving, he states that students may need guidance to discover and productively make sense of new knowledge. In particular, having learnt Logo programming extensively under pure discovery conditions, students struggled with fundamental code constructs, had misconceptions about programmatic logic and remained at the novice level in planning (Mayer, 2004).

Studies utilizing contemporary programming environments have also shown that students can face and require help with challenges regarding specific computational concepts (e.g. Franklin et al., 2013). Programming will likely always result in errors that are typically caused by a discrepancy between the programmer's mental model (what they wish to achieve) and what the project actually does (Ben-Ari, 1998). A well-known practice, particularly in the context of more contemporary graphical programming contexts, is 'bricolage'. This is described by Ben-Ari (1998) as 'endless debugging of the "try-it-and-see-what-happens" variety'. Additionally, programmers can encounter more programming practice centered challenges regarding, for instance, shared knowledge creation in pairs (e.g. Lewis & Shah, 2015).

Alongside even more distant research on school students' constructionist learning in programming (e.g. Suomala, 1999), this study understands that discovery can be a main building block in learning in the spirit of constructionism, but it should not leave learners alone to rely on chance with their potentially unproductive and even misleading knowledge construction strategies. Instructors can facilitate and guide learning, for instance, by organising learning contents, employing cognitive apprenticeship, assisting with cognitive conflicts and providing scaffolding (e.g. Collins et al., 2018). On a general level, Lee et al. (2011) coined the 'use-modify-create' model to define an overall progression for learning programming. Similarly, Xie et al. (2019) proposed that novice programmers learn to trace code, write syntax, comprehend generalisable computational abstractions and implement those abstractions incrementally. Carlborg et al. (2019) proposed a scope of autonomy model to support teachers in developing appropriately and incrementally challenging instructional material for programming. In a similar fashion, Franklin et al. (2020a) discovered several behavioural improvements with a learning strategy called TIPP&SEE, which has students examine premade projects and then explore their contents. Weintrop et al. (2019) developed a rubric to assist teachers in evaluating computing curricula, and Coenraad et al. (2020) confirmed its effectiveness with teachers.

On a more grassroots level, several pedagogical methods and strategies, such as using information processing activities (e.g. metaphors, mind mapping)

and supporting meta-cognitive processes via reflection, have been employed to support students' learning in programming education at different educational levels and for different phases in teaching and learning processes (Garneli et al., 2015; Hsu et al., 2018; Lye & Koh, 2014; Manches et al., 2020; Webb & Rosson, 2013). In ensuring that students are working to produce meaningful knowledge, they can be provided solution hints, clarification of concepts, milestones for their processes and scaffolding for learning specific skills, such as using computational patterns used in programming games (Luo, 2005; Nickerson et al., 2015). Scaffolding can also be used, for example, to help in breaking the program down into smaller programs (e.g. how to program dialogues between sprites) (Lye & Koh, 2014) or elucidating what contents are required to design specific creative features, such as score counting in a game. A contemporary instructional element can also be the connected nature of learning where expertise can be networked. Students can participate in communal knowledge-creation processes by engaging with their peers and other programmers through various resources on the Internet, especially by searching and remixing existing materials on the Scratch website (Brennan & Resnick, 2012).

Perhaps more fundamentally, students can program in pairs to achieve a common goal so that one student operates as the 'driver' (controlling the computer) and the other operates as the 'navigator' (assisting in reviewing and verifying the design) (Höfer, 2008). Pair programming has generally been confirmed to produce several advantages, such as improvements in the quality of design, enhanced knowledge creation, reducing errors and defects in the work and improving novice programmers' motivation in particular (Arisholm et al., 2007; Preston, 2005; Wei et al., 2021). Such advantages can be theoretically affiliated with principles in social constructivist learning, such as negotiation of different viewpoints (Roschelle & Teasley, 1995). Although research-based knowledge of collaboration is thin in this educational context, collaborative programming has generally been confirmed to be effective despite the fact that it can add further components that burden the already demanding problem-solving process (Scherer et al., 2020). In fact, success in pair programming has been generally shown to be influenced by students' skill levels, previous learning histories, learning strategies, attitudes, personalities, emotions and even physical environments (Ally et al., 2005; Denner et al., 2014; Campe et al., 2020; Scherer et al., 2018).

In summary, there is significant pedagogical knowledge regarding students' learning in programming, but key information about CT learning during pair programming in Scratch in naturalistic classroom situations is scant and disordered. In other words, effective teaching and learning practices for CT in contemporary programming contexts are not yet fully understood as a tangible and practically applicable pedagogical system. One reason is that CT is multifaceted, and previous studies have viewed it in discrepant ways and have mainly focused on second-order programming concepts rather than first-order cognitive CT as viewed through, for instance, its core educational principles (see subsection 2.1.3). Another reason may be that programming is a new topic in terms of the utilised technologies (e.g. networked programming environments) and contemporary

pedagogical (e.g. collaborative learning) and epistemological underpinnings (e.g. constructionism). Additional empirical information from classroom situations is needed to enrich evidence-based knowledge of teaching and learning CT in Scratch. Additionally, a distinct theoretical framing could be utilised to organise key information about students' CT learning and ways to support it. In this study, such a framing is as follows.

#### **2.2.4 Assessment for learning**

An outlying goal of this thesis is to develop ways to support students' CT learning through Scratch in school classrooms. An important motivation for selecting this viewpoint stems from the fact that programming is often difficult for students even when employing easy-to-use graphical tools (Lye & Koh, 2014). This study ties knowledge about students' CT learning at the primary school level and ways to support it in the context of Scratch to the pedagogical interest of 'assessment for learning'. In concrete terms, this interest aligns with the practice of 'formative assessment'.

In terms of background, CT is multifaceted; it involves, among other things, an understanding of computational concepts and practical skills in problem solving, particular vocabulary and certain worldviews, attitudes and characteristics (Barr & Stephenson, 2011; Csizmadia et al., 2017; Grover & Pea, 2018; Lonka et al., 2018; Shute et al., 2017). The complex nature of CT has resulted in a need to move towards holistic systems of assessment that probe it from multiple entry points (Basso et al., 2018; Grover et al., 2017). Different types of assessments can contribute to holistic assessment systems in CT for measuring, documenting and supporting student achievement (Basso et al., 2018). Although no established holistic assessment systems exist, several previous works provide methods that could contribute to their creation. In fact, the assessment of programming, which is a central pathway to develop CT, goes back decades (e.g. Pea et al., 1987). There are also several relatively recent literature reviews on ways to assess CT (Cutumisu et al., 2019; Da Cruz Alves et al., 2019; Lockwood & Mooney, 2018; Roman-González et al., 2019; Shute et al., 2017; Tang et al., 2020). Taken together, the reviews reveal that perhaps more than a hundred unique assessments of CT through programming currently exist.

As an overview, specific assessments in CT have targeted different educational levels ranging from primary schools to university training and in-service teacher training. In particular, assessment set in the context of formal education has begun to claim a significant share of research, especially in recent years. Students' skills and understanding regarding CT have been assessed both generically and in practical contexts, such as block-based programming languages (e.g. Scratch, Alice), web-based simulations, robotics and game-based tasks. Assessments have been implemented in authentic programming scenarios, with portfolios and traditional methods, such as interviews, self-report scales and surveys. They have been mainly computer-based, and few have been automatised. A common approach has been to develop an assessment for CT based on the particular

theoretical viewpoint of CT adopted. Most studies have been conducted in the US (Cutumisu et al., 2019; Shute et al., 2017; Tang et al., 2020).

One way to make sense of the myriad of assessments in CT is to consider the purpose of the assessment. For example, the various assessment approaches can be distinguished based on whether they are intended to be used by the teacher, the student or an administrator (Da Cruz Alves et al., 2019). From this viewpoint, three types and purposes of assessment can be distinguished: *diagnostic*, which aims to identify students' preconceptions, lines of reasoning and learning difficulties to inform teachers about students' existing knowledge; *summative*, which aims to measure and document learning primarily for the purpose of grading; and *formative*, which aims to inform instruction and provide feedback to students on their learning (Black & Wiliam, 1998; Keeney, 2008).

In one sense, CT can be perceived as a quantifiable product of learning, that is, acquired mastery that can be measured for summative or diagnostic purposes (Tang et al., 2020). Examples include summative tests measuring scholastic aptitude, skill-transfer assessments, perception and attitude assessments with questionnaires and vocabulary tests (Roman-González et al., 2019). Such methods have measured, for instance, students' skills in CT or programming and their dispositions and attitudes toward CT (Tang et al., 2020). Notable examples include the Bebras task, which measures the transfer of CT to various types of real-world problems (Dagienè & Futschek, 2008). Román-González et al. (2017a) developed and validated a CT test measuring students' abilities regarding such second-order programming concepts as sequences, loops and conditional structures. Zapata-Cáceres et al. (2020) adapted this test for early educational stages. A notable large-scale comparative assessment is conducted in the ICILS study (Frailon et al., 2020). Moreover, focusing on a non-cognitive domain in CT, Kong et al. (2018) and Mannila et al. (2020) developed survey instruments to investigate such matters as students' self-efficacy, interest and attitude toward programming.

CT can also be assessed as a learning experience or a process, which can be difficult with all traditional assessment methods (Tang et al., 2020). Rather than being generic, the assessment can connect to a particular learning situation, for example, by employing an automated script, an application or manual rubrics to evaluate concrete CT-related learning (e.g. a programming project or a process). Such tools and rubrics exist and can be used to effectuate either static, dynamic or manual analysis to provide dichotomous, polytomous or composite scoring of work (Da Cruz Alves et al., 2019). Notable examples include the Fairy Assessment (Werner et al., 2012), which evaluates students' performance in terms of comprehending, designing and solving pre-provided programming tasks in the Alice programming environment. Repenning et al. (2015) developed Computational Thinking Pattern Analysis for measuring and visualising how students implemented CT-related elements in games they programmed in the AgentSheets programming environment. In a similar work, Koh et al. (2010) developed the Computational Thinking Pattern graph that evaluates students' game projects made with the AgentSheets software and provides visual feedback regarding key patterns (e.g. cursor control, collision) implemented in the games. Koh et al. (2014)

developed the Real Time Evaluation and Assessment of Computational Thinking framework for the Scalable Game Design Arcade programming environment to provide teachers a dashboard to evaluate their students' proficiency in the above-mentioned patterns.

Many of the tools or rubrics assessing CT as a learning experience or a process are intended for summative use, as only a few of them provide explicit suggestions or tips on how to improve the design (Da Cruz Alves et al., 2019). Assessments that aim to enhance students' learning have been altogether overlooked in previous studies, and future developments in CT assessment are encouraged to be more learner-driven, to identify individual learning paths and gaps and to integrate instructional feedback (Basso et al., 2018; Robles et al., 2018). Assessments could therefore utilise, for example, tools that can provide feedback in terms of individual progress in set learning goals (Roman-González et al., 2019), which is more strongly linked with the core purpose of assessment for learning. Educators could benefit greatly from methods that can support and guide students' learning instead of methods that merely establish 'where the students are' (Brennan & Resnick, 2012; Lye & Koh, 2014). Normative rather than individual and mastery-oriented assessment approaches can also promote inadvertent competition among students and can cause students with more difficulties in learning to become demotivated and lose confidence in their capacity to learn. Instead, classroom-based tasks associated with formative assessment can enhance students' individual learning performance and their belief about their own performance capacities (Black & Wiliam, 1998; 2009).

Two foundational ideas are at the core of formative assessment: a student's perception of a gap between a desired learning goal and their present skill or understanding and action taken by the student to close that gap (Black & Wiliam, 1998). Black and Wiliam (1998; 2009) concretised instructional processes that can promote the effectuation of such underpinnings in classroom situations. In short, students are presented learning goals that they understand. Their subsequent learning work is supported by activating them as instructional resources for one another and guiding them to understand how their current skills differ from the set learning goals. The students are then guided to take appropriate action to match their skills with the learning goals (Black & Wiliam, 1998; 2009). Such a framing could be used to encapsulate and organise relevant pedagogical information, such as effective instructional strategies and guidance methods for CT education in Scratch in classrooms. This study adopts this framing to examine students' CT in Scratch from two viewpoints justified by the nature of the learning context at hand: assessing the 'what' (the programming contents in students' Scratch projects) and the 'how' (students' pair programming activities).

### **2.2.5 Assessing programming contents and activities**

Previous research has carried out multiple analyses concerning programming contents (the 'what') and programming activities (the 'how') in Scratch through versatile operational methods and theoretical connections to CT. In other words, existing assessments vary greatly in terms of how the assessment has been

developed based on a particular theoretical viewpoint of CT (Shute et al., 2017). Illustratively, the developers of Scratch found that CT manifests as ‘concepts’ (e.g. sequences, loops), ‘practices’ (e.g. iteration, remixing) and ‘perspectives’ (e.g. questioning, expressing) in the programming environment in question (Brennan & Resnick 2012).

It is important to return to the distinction between ‘first-order cognitive CT’ and ‘second-order programming concepts’, however (see subsection 2.1.2). CT encompasses various skills that are expected to be transferable across computational contexts. Therefore, this thesis views CT as being represented by cross-contextual (i.e. first-order) key concepts and practices, such as ‘algorithms’, ‘abstraction’ and ‘problem decomposition’, and more specific skills and knowledge concretised through the core educational principles (see subsection 2.1.3). In turn, various programming contents and programming activities (i.e. second-order) may be relevant in displaying and fostering CT. To investigate CT through programming contents and activities sufficiently and support the journey to educational practice, it is necessary to clarify what second-order programmatic affordances contextualise the different first-order CEPs<sup>14</sup>. Figure 10 illustrates this reciprocal theory–practice space and highlights the primary objective of Article I: contextualising the multifaceted theory of CT to the versatile programmatic practice in Scratch.

The quest for contextualisation appears promising in terms of the richness of both the first-order and second-order dimensions. CT has rich existing descriptions, and the programming contents and activities in Scratch have been operationalised in several studies although typically without a clear connection to specific skill or knowledge areas in CT.

Assessment of the ‘what’, that is, the programming contents (e.g. the code) in students’ Scratch projects could be viewed as a form of content analysis based on observed data in programmed artefacts (‘artefact analysis’). Students’ programming projects are rich, concrete and contextualised examples and demonstrations of their conceptual encounters with CT (Brennan & Resnick, 2012; Grover & Pea, 2013). The indications of CT in creatively designed projects can reveal the progression of students’ conceptual understanding and application of CT. This can be revealed using formative assessment methods, such as examining students’ projects and portfolios of projects to evince their development in computational design over time (Brennan & Resnick, 2012).

Artefact analysis has been widely adopted in estimating students’ CT in Scratch in previous research. For instance, Maloney et al. (2008) examined the frequency of particular code constructs (second-order programming concepts) in students’ Scratch projects. Franklin et al. (2013; 2017) and Meerbaum-Salant et al. (2013) examined similar contents in students’ projects more qualitatively (e.g. how they were implemented). Moreno-León et al. (2015) developed ‘Dr. Scratch’,

---

<sup>14</sup> It is important to note, however, that some conceptual and practical dimensions in CT as viewed herein through the CEPs (see subsection 2.1.2) may stem – at least partly – from programming methodology rather than the other way around as a rule. Therefore, the directions of fit in Figure 10 are not necessarily always purely one-way.

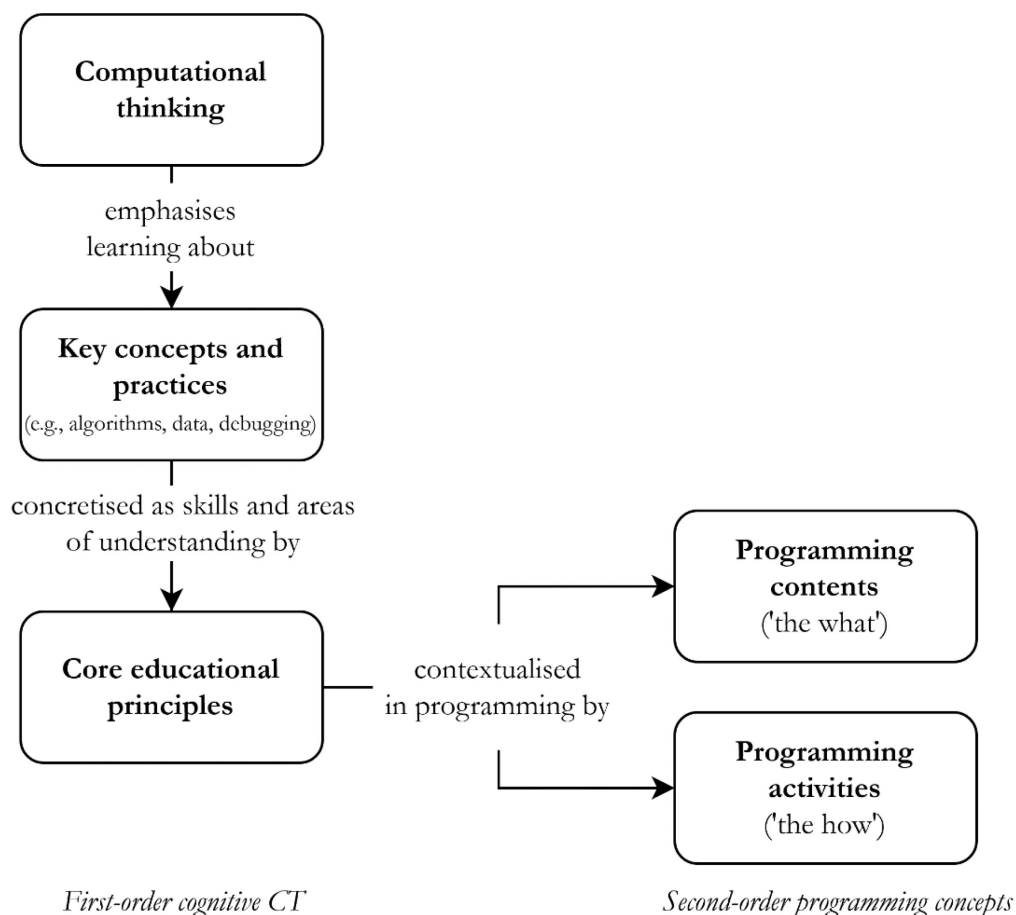


Figure 10. The relationship of CT and programming as interpreted in this study

an automated assessment framework, to reveal the computational sophistication of Scratch projects based on the presence of particular code constructs. Seiter (2013) investigated how students performed when solving pre-designed learning scenarios. Seiter and Foreman (2013) examined so-called ‘design pattern variables’, that is, semantically meaningful combinations of code constructs that established creative features, such as animations and user interaction, in students’ projects as indications of their CT. Other studies (Basu 2019; Funke et al., 2017; Wilson et al., 2012) examined what kinds of non-programmatic contents, such as instructions to use the project and the appropriate naming of the designed computational objects, students’ projects included.

In turn, students’ programming processes can denote ‘practical encounters with CT’, going hand-in-hand with projects they have designed and providing a complementary view of students’ understanding of concepts and skills in the practices they employ (Brennan & Resnick, 2012). Some aspects in CT practices can be social (e.g. collaboration) or highly process-oriented (e.g. iterative development), and they can become evident in temporal examinations of programming processes or via social interactions. In fact, in programming contexts other than Scratch, examinations of programming processes rather than of the final

projects have uncovered ‘counterintuitive data’ and ‘patterns with better predictive power than exams’ (Blikstein et al., 2014). In Scratch, previous studies have explored what kinds of design phases students’ programming processes involve (Burke, 2012), to what extent students have carried out various tasks, such as analysing scripts or testing play (Ke, 2013), and how students talk and operate in programming roles at different times (Lewis, 2011; Lewis & Shah, 2015; Shah et al., 2014; Tsan et al., 2018). Previous research has shown especially that the quality of talk may vary greatly during pair programming and that a risk for one-sided decision-making in favour of the driver can be high (Deitrick et al., 2014; Tsan et al., 2020; Zakaria et al., 2019).

This thesis focuses on ways in which assessment could benefit students’ learning in authentic programming situations. Therefore, after contextualising relevant CT-related contents and activities to teach and learn in Scratch, an opportunity for operationalising them for enacting formative assessment processes (setting learning goals, evincing learning, and providing feedback) emerges. Previous studies have not thoroughly considered ways to effectuate those processes in Scratch or other programming environments. As an example, currently available tools for automated assessment of projects in Scratch, such as ‘Dr. Scratch’ (Moreno-León et al., 2015) and ‘Ninja Code Village’ (Ota et al., 2016), assess CT generically instead of contextually. Unlike Black and Wiliam (1998) suggest as a way of promoting formative assessment, they provide feedback related to absolute levels of performance rather than progress. Nonetheless, the effectuation of formative assessment in Scratch appears conceivable. By means of assessing students’ programming projects and programming processes, the empirical sub-studies in this thesis (Articles II and III) aim to attain rich evidence concerning how the teaching and learning of CT can occur in Scratch in primary school classrooms. The pedagogical relevance of this evidence is promoted by discussing it in terms of the processes portrayed in formative assessment in a theoretical framing described as follows.

As programming can be reasoned to foster students’ CT (Brennan & Resnick, 2012), learning goals for CT can be established indirectly in the form of programming contents to manipulate and programming activities to carry out in Scratch. Scratch programming contents and programming activities recognised in previous literature could aggregately cover CT comprehensively and could be formulated as creative features in Scratch (e.g. animations, user interaction) to design in games or animations to advocate for their relevance for students. Similarly, evidence that demonstrates students’ CT capabilities and progressions could be gathered by assessing what they have programmed in their Scratch projects and how they manage to effectuate programming activities (Grover & Pea, 2013; Román-González et al., 2019; Seiter and Foreman, 2013). Perhaps more precisely, the contents that students have implemented in their projects can indicate ‘conceptual encounters’ they have had with conceptual aspects in CT (e.g. data, algorithms), and the programming activities that students carry out can indicate ‘practical encounters’ they have had with more practical aspects in CT (e.g. iterative design, debugging).



On the other hand, it is imperative that the evidence of learning generated and displayed for assessment is relevant (Black and Wiliam, 1998). There are known risks involved with block-based programming environments in particular in which students can make design decisions without knowing exactly what they are doing. It is therefore vital to consider what students are thinking while implementing computational designs (Lye & Koh, 2014). Ways to do so are assessing primarily semantically rather than merely technically meaningful programming contents in artefact analysis (Seiter & Foreman, 2013) and examining how the projects came into fruition through a particular kind of design process (Lye & Koh, 2014). It may also be relevant to consider the context of the programming activity (e.g. the type of project, programming objective) (Moreno-León et al., 2017) and its determining nature (e.g. more open-ended design or working with pre-existing materials) (Lee et al., 2011).

Last, when assessment takes place during actual programming processes, targeted feedback could be provided for key places (e.g. scripts) in the programming project being designed or the ongoing programming activities at appropriate times (Lye & Koh, 2014). For instance, meaningful and authentic feedback can be provided with respect to errors or bugs in the project (Hao et al., 2021; Vi-havainen et al., 2013) or an ineffective programming strategy, such as unproductive trial-and-error programming (see Ben-Ari, 1998).

### 3 AIMS AND RESEARCH QUESTIONS

The lack of comprehensive and well-established theoretical models for CT through programming learning activities and the scarcity and disorganised state of empirical evidence of students' learning from authentic school classroom environments has made research-based pedagogical decision making troublesome. Despite the global interest in introducing CT in schools, several matters have remained unclear. These include knowledge regarding what is the more general-level (cf. an overly programming-centric) educational goal for introducing CT in primary education, what conceptual and practical areas in CT students can learn and how students' CT learning can be supported while they are using such age-appropriate and popular programming environments as Scratch collaboratively in authentic classrooms.

To recap, the broad aims of this thesis are to shed light on how the learning of multifaceted CT can be supported by assessing students' Scratch programming in primary school classrooms in terms of teaching, learning and assessing CT through programming with Scratch. The actions taken in this thesis (see Figure 3) to meet its aims can be shaped as a research procedure in which answers to two research questions (RQs) are sought.

The first RQ is:

- 1) How have the skill and knowledge areas affiliated with multifaceted CT been assessed in Scratch at the primary school level?

This evaluative question stems from an initial overview of background literature performed at the beginning of this research. Subsection 2.1.2 underlines the theoretical complexity surrounding the term 'computational thinking' and highlights operational inconsistency concerning what kinds of core cross-contextual skills and knowledge it can be framed to encompass. As a consequence, no clearly articulated educational objective exists for learning CT comprehensively (i.e. what to teach and learn in it) through programming in primary education. Therefore, as a theoretical background, CT was concretised in the context of programming in subsection 2.1.3. However, as this concretisation was novel, there is no

systematic categorisation depicting how the manifold practical affordances in specific programming environments (namely Scratch) can foster students' more general-level CT. Therefore, previous studies focusing on assessment in Scratch at the primary school level need to be reviewed next to this new perspective to generate theoretical and operational underpinnings for further empirical research in this study. Article I responds to this need by employing a systematic literature review.

The second RQ is:

- 2) How did 4<sup>th</sup> grade students encounter CT conceptually and practically while programming with Scratch in general classrooms?

This question demarcates an empirical investigation that stems from the theoretical and operational background work done for RQ1. In particular, subsection 2.2.3 overviews the lack and disorganised state of evidence-based knowledge and methods for assessing how students can learn the various conceptual and practical areas involved with CT in Scratch in classrooms, thus problematising pedagogical decision making, such as ways to support students' CT learning. Therefore, this RQ operates indirectly as a pathway for developing new methods to assess students' CT. However, its core purpose is to employ those methods to attain rich insight regarding how 4<sup>th</sup> grade students were involved with CT during a programming course in naturalistic classroom situations. Article II focuses on this matter by assessing the students' programmed Scratch projects (artefact analysis), and Article III focuses on the matter by examining students' pair programming processes (process analysis).

## 4 CONTEXT AND DESIGN

This chapter describes the context and design of the study. It includes the educational context, that is, Finnish primary education and especially programming education positioned mainly under the transversal competence ‘ICT competence’ in the primary school core curriculum (in section 4.1). The chapter also provides information about the students who participated in data collection (in section 4.2) in addition to the programming course in which they participated and the data that was collected in this research (in section 4.3). The main underpinnings of analysis of the collected data are subsequently outlined (in section 4.4), and the epistemological underpinnings regarding the research design (in section 4.5) and ethical aspects considered in conducting this study (in section 4.6) are elaborated.

### 4.1 Primary education in Finland

#### 4.1.1 The core curriculum

After compulsory preschool education, Finnish students attend a 9-year compulsory comprehensive school that starts at age seven and finishes at age 16. Nearly all primary schools are public and are funded and administered by municipalities. All schools follow national curriculum guidelines, which are rebranded and published every 10 years, but teachers at the grass-roots level have a relatively significant amount of autonomy in deciding upon, for example, the methods of instruction. Schools and/or municipalities are also permitted to devise their own local curricula provided they do not conflict with contents in the national standard core curriculum (Opetushallitus, 2014).

In August 2016, the Finnish primary school system adopted the latest core curriculum (Opetushallitus, 2014), which remains in effect until 2026. Fundamental overall pedagogical premises embracing the core curriculum include that students are encouraged to be increasingly active, autonomous and cooperative goal

setters and problem solvers. In practice, learning activities in schools should involve knowledge building through active thinking and doing, investigating various real-life phenomena and planning and reflecting on one's own work alone and together with others. Learning is also viewed as pervasive in various contexts and in all stages of life (Opetushallitus, 2014, p. 17).

Compared to the previous curricula, methods for assessment are more diverse in the new curriculum. The main purpose of assessment in the curriculum is to support the students' learning by providing them versatile and frequent feedback on their learning progress. Additionally, students' capability to assess their own and their peers' learning is expected to be developed. In practice, students are encouraged to participate in the assessment of their own learning by understanding and discussing assessment criteria and their progression regarding them. Assessment always rests upon set learning objectives and criteria. The core curriculum comprises concrete assessment criteria signifying 'good performance' (i.e. mark '8' on a scale of 4 to 10) in every subject at the end of the 6<sup>th</sup> and 9<sup>th</sup> grades. Each teacher assesses their students by referring to the mark '8' competence descriptions in the core curriculum (Opetushallitus, 2014, p. 47–54).

For the first time in the Finnish school system, the core curriculum also explicitly characterises seven so-called 'transversal competences'<sup>15</sup>, which are sketched as broad skill, knowledge, value, attitude and volition areas that penetrate all curricular areas (and each other). An expressed rationale of the transversal competences is that knowledge is holistic, and practical activities in life require skills that transcend and combine different ways of thinking and doing. Real-world phenomena thus encompass connections and interrelationships that can be meaningfully explored by examining larger wholes instead of bits and pieces of facts. Although teaching and learning in the context of traditional subjects still takes place, students are encouraged to engage in authentic and interdisciplinary learning in theme-based or phenomenon-based settings that involve conceptual or practical elements from more than one traditional school subject. A concrete requirement in the core curriculum is that schools are obligated to organise at least one multidisciplinary learning module with a clearly defined theme each semester (Opetushallitus, 2014, p. 20; 31–32).

#### 4.1.2 ICT competence (T5)

A key revision in the new Finnish primary school core curriculum is the ubiquitous role of ICT in teaching and learning via the fifth transversal competence, ICT competence (T5), which is introduced in the curriculum in the following manner:

ICT competence is an important civic competence in itself and as a part of multiliteracy. It is both a learning objective and a tool for learning. All students should have the opportunity to develop their ICT skills during primary education. ICT is employed in

---

<sup>15</sup> The seven transversal competences in core curriculum are: (1) Thinking and learning to learn; (2) Taking care of oneself and others, managing daily activities, safety; (3) Cultural competence, interaction and expression; (4) Multiliteracy; (5) ICT competence; (6) Competence for the world of work, entrepreneurship; (7) Participation and influence, building the sustainable future (Opetushallitus, 2014).

an organised fashion during all primary grades, in different subjects, and in multidisciplinary learning modules and other schoolwork. (Opetushallitus, 2014., p. 23.)

The four main domains of ICT competence are as follows:

- 1) Students are guided to understand the operational and functional principles of ICT in addition to the fundamental concepts and develop their practical ICT skills while making their own creations.
- 2) Students are instructed to use ICT responsibly, safely and ergonomically.
- 3) Students are taught to use ICT in information management and in inquiry and creative work.
- 4) Students gain experiences and practice ICT use for interacting and networking. (Opetushallitus, 2014., p. 23.)

ICT competence is grounded in the surrounding world in the following way:

Students are guided to recognise different applications and purposes of ICT and notice their meaning in everyday life, in interactions between people and as means to influence. Why ICT is required in studying, work and the society and how these skills have become a part of general working life skills are pondered together. Students learn to evaluate the effect of ICT in terms of sustainable development and act as responsible consumers. During primary education, students gain experiences in using ICT also in international interaction. They learn to perceive its meaning, opportunities, and risks in the global world. (Opetushallitus, 2014., p. 23.)

### 4.1.3 Programming in Finnish schools

Although programming is not an unprecedented educational topic in primary education, it is explicitly mentioned as a mandatory learning content in the core curriculum for the first time in Finnish education in three settings<sup>16</sup> – ICT competence (T5) and the subjects of math and crafts. The specific content areas, teaching objectives and assessment criteria of programming in the curriculum are divided into three grade categories – grades 1-2, 3-6 and 7-9.

Chronologically, programming is first positioned under grades 1-2 in the following ways:

**ICT competence (T5):** Students gain and share experiences in working with digital media and age-appropriate programming (Opetushallitus, 2014, p. 101).

**Math (teaching objective):** To train the student to form sequences of instructions and act according to instructions.

**Math (content area):** Becoming familiar with the basics of programming starts by forming sequences of instructions, which are also tested (Opetushallitus, 2014, p. 129).

---

<sup>16</sup> Programming and CT are in all likelihood new professional topics to most teachers who had not received formal training in them in pre-service teacher training prior to the newest core curriculum. Systematic in-service support (e.g. professional training programs) was not provided by educational policy organizations either while the curricular change was taking place.

For grades 3–6:

**ICT competence (T5):** As students try programming, they gain experiences regarding how the functionality of technology depends on human-made solutions (Opetushallitus, 2014, p. 157).

**Math (teaching objective):** To inspire the student to form sets of instructions as computer programs in a graphical programming environment.

**Math (content area):** Planning and implementing programs in a graphical programming environment (Opetushallitus, 2014, p. 235).

**Math (mark ‘8’ when finishing the 6<sup>th</sup> grade):** The student should be able to program a functional program in a graphical programming environment (Opetushallitus, 2014, 239).

**Crafts (content area):** Practicing the functionalities established with programming, such as robotics and automation (Opetushallitus, 2014, p. 271).

For grades 7–9:

**ICT competence (T5):** Programming is practised as a part of the studies in different subjects (Opetushallitus, 2014, p. 284)

**Math (teaching objective):** To instruct the student to develop their algorithmic thinking and skills to apply mathematics and programming in solving problems.

**Math (content area):** Programming and simultaneously practicing good programming practices. (Opetushallitus, 2014, 375.)

**Math (mark ‘8’ when finishing the 9<sup>th</sup> grade):** student should be able to apply the principles of algorithmic thinking and to program simple programs (Opetushallitus, 2014, p. 379).

**Crafts (content area):** Using embedded systems in crafts, that is, applying programming in plans and manufacturing products (Opetushallitus, 2014, p. 431).

## 4.2 Participants

Students between grades 4 and 6 were initially sought to participate in the study. Motivated teachers at a local professional coding-themed training event (the ‘coding roadshow’ event organised by the Innokas Network<sup>17</sup> in 2016) were recruited via an open, verbally expressed invitation. An incentive for participation was the opportunity for a visiting teacher from the university to assist in introducing programming to the students in the teachers’ classrooms.

---

<sup>17</sup> The Innokas network is a network of teachers, learners, school leaders and other educational experts mainly in Finland. The network organises free of charge professional training and events aiming to educate active 21<sup>st</sup> century learners, especially from the viewpoint of versatile use of technology (<https://www.innokas.fi/en/>).

Three teachers showed interest in participating, and one 4<sup>th</sup> grade teacher with a convenient schedule was selected to participate in the study. The teacher represented two of their colleagues in an average-sized Finnish municipal primary school in a medium-sized town, effectively allowing a group of three 4<sup>th</sup> grade classes ( $C_a$ ,  $C_b$ ,  $C_c$ ) and their regular teachers ( $T_a$ ,  $T_b$ ,  $T_c$ ) to participate in the study. The three classes comprised 22 ( $C_a$ ), 21 ( $C_b$ ) and 26 ( $C_c$ ) students, of which 57 (62% girls, 38% boys) acquired the informed consent of their legal guardians. The students ( $S_1 \dots S_{57}$ ) were 10–11 years old when the study started. The students were surveyed and found to be generally inexperienced in programming and were thus found to represent a typical general classroom. Only a few students had had previous programming experience:

‘I have programmed with Scratch Jr. and in Code[.org] website’ ( $S_1$ )

‘With dad’ ( $S_{13}$ )

‘Code.org’ ( $S_{17}$ )

‘I have made three games with Scratch’ ( $S_{21}$ )

‘It was some Angry birds’ ( $S_{22}$ )

‘Scratch Junior’ ( $S_{41}$ )

‘Can’t remember its name but sometimes I edit with iMovie’ ( $S_{45}$ )

‘Little bit with a program whose name I don’t remember’ ( $S_{47}$ )

Two of the 57 participating students were non-native Finnish speakers. The classes also included students with special needs who participated in the programming activities but did not choose to participate in the data collection.

Each of the three classes attended a programming course separately once per week for approximately 4 months (12 sessions in total) in 2017. Due to the participants’ young age and their novice stage of learning in programming, the primary objective of the course was to introduce the students to fundamental Scratch features and CT through perceivably introductory programming contents and programming activities. Learning was intended to occur in the context of interest-driven design of creative media projects, such as interactive animations and stories, in programming in pairs.

The design of the course was inspired by previous studies utilising Scratch (Grover et al., 2014; Meerbaum-Salant et al., 2013). Learning activities during the course encompassed unplugged exercises, more and less guided and discovery-focused tasks, debugging challenges, worked examples for remixing and more open-ended or thematised design projects. The employed learning materials included selections from the *Creative Computing* guide (Brennan et al., 2014). The implementation of the programming course and the data collection were piloted in one 3<sup>rd</sup> grade classroom in another school prior to the actual programming course and data collection.



The author of this thesis ('visiting teacher') operated as the primary instructor of the course. The regular teacher of each class was always present. A research assistant and a learning assistant for the special needs students were present during most sessions. All the teachers guided the students' work. Sharing, informal collaboration with peers and working at home were encouraged.

All sessions apart from the unplugged exercises (session 1, see sub-section 4.3) took place in the school's computer lab. The lab had 15 computers available for the students to use. The regular teachers grouped the students (two or three students per group) based on perceived shared skill level or similar interest areas. Group compositions remained primarily the same throughout the course, although sporadic variations were made due to absences or cooperative difficulties among the students. Only a few students worked alone either for the entire course or during certain sessions.

### 4.3 Data collection

The data collected and selected for further analysis included a corpus of previous studies (RQ1/the literature review) and two datasets collected from the participants (RQ2/the empirical analyses). The data for the literature review is described in more detail under the summary of Article I (in subsection 5.1). The empirical datasets included the students' Scratch projects made during the sessions and video recordings of students programming in pairs while programming their final projects of the course.

In terms of the Scratch projects, the students programmed different kinds of 'design', 'tutorial', 'remix' and 'debug' projects during the course (Table 1). The tutorial projects (P2), debugging challenges (P4, P7) and remixed projects (P5) involved pre-set objectives that guided towards modifying specific programming contents. Typically, sessions focusing on these projects began with a teacher-led demonstration of a particular computational or creative feature (e.g. sprites sprint racing) or an incomplete Scratch project that required implementing or debugging particular programming contents. The students were subsequently instructed to follow the tutorial or remix and finalise and creatively extend their own projects. In contrast, the programmatic requirements of the design projects (P1, P3, P6, P8) were less rigid. Typically, the students imagined and designed their own projects within certain boundaries, having an opportunity to seek information from the Internet. With projects P3 and P8, the students were able to plan their projects with pen and paper over one prior session. In addition to collecting the finished projects, the students' written plans for the final projects (P8) were photographed when programming of said projects started.

Once the course ended, all projects that the students had returned in 'Scratch studios' that were created for them were collected as data. Projects made outside the sessions were excluded because they were found to be mainly incomplete drafts. Authorship of the projects was determined based on two criteria:

Table 1. Scratch projects made weekly during the course

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, 'initialisation', '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

(\*) Programming processes were video and audio recorded, and hand-made plans for the projects were photographed.

the student had returned a self-reflection sheet from the session (confirming attendance), and the student's name was written on the project introduction page (confirming participation in design). Both tasks were enforced by the teachers at the end of each session.

As the second main empirical dataset, the programming processes of 12 random pairs (dyads), four dyads from each of the three classes, were video and audio recorded when they programmed their final projects (P8) over two 45-minute sessions. The computer screens were recorded with software called CamStudio, and a GoPro camera and a voice recorder were set next to each dyad (see Figure 11). CamStudio was used to simultaneously record each group's on-screen activity on the computer. To enhance the students' voice levels, a digital voice recorder was placed under the monitor of the computers. The raw audio and video data streams were synchronised with video editing software to produce full session video files in which the dyads' talk and behaviour on and off the computer while programming their projects transpired. Dyads with absentees and those with data losses due to technical difficulties were omitted, resulting in

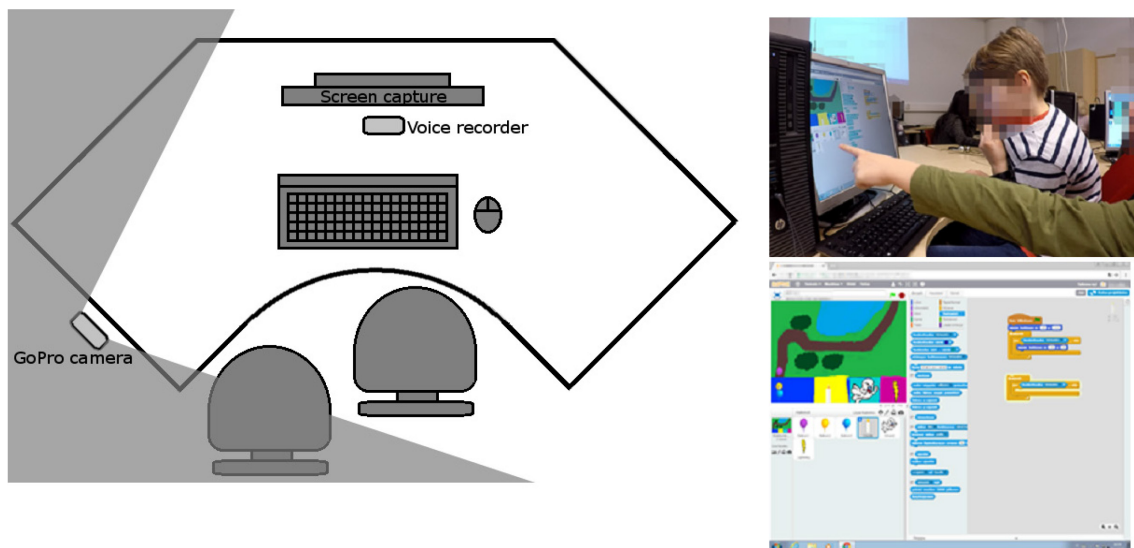


Figure 11. The data collection setting and sample footage for Article III

four dyads with rich data for analysis. In total, eight videos (mean duration = 38 min, 0 s) – two sessions from the four dyads – were analysed. The dyads’ initial project plans and screen captures of their finished projects in Scratch were also collected as data.

#### 4.4 Data analysis

The mixed methods analysis of the data in this thesis split into two approaches: theoretical analysis for RQ1 and empirical analysis for RQ2 (Figure 12). The analysis performed for RQ1 intended to specify the educational goals of CT and evaluate ways to assess CT in Scratch (the first two main actions of this thesis, see Figure 3). Theoretically, the analysis leaned on the distinction between ‘first-order cognitive CT’ – construed herein via the CEPs – and ‘second-order programming concepts’ (‘the what’ and ‘the how’ in Scratch programming) (see Figure 10). In concrete terms, the analysis sought to contextualise the theory of the cross-contextual CEPs to the programmatic practice in Scratch. As overviewed in chapter 2, CT has rich existing descriptions, and the programming contents and activities in Scratch have been operationalised in several previous studies. Thus, extensive literature searches were seen as a promising pathway to systematically gather and re-structure existing information in this educational setting.

The analyses performed for RQ2 intended to develop new assessment methods for CT in Scratch and provide rich empirical insight about learning in Scratch at the primary school level (the third and fourth main actions of this thesis, see Figure 3). The analyses built on the theoretical framings established as results of the literature review: the gathered and re-structured operational measures for assessing students’ CT based on Scratch programming contents

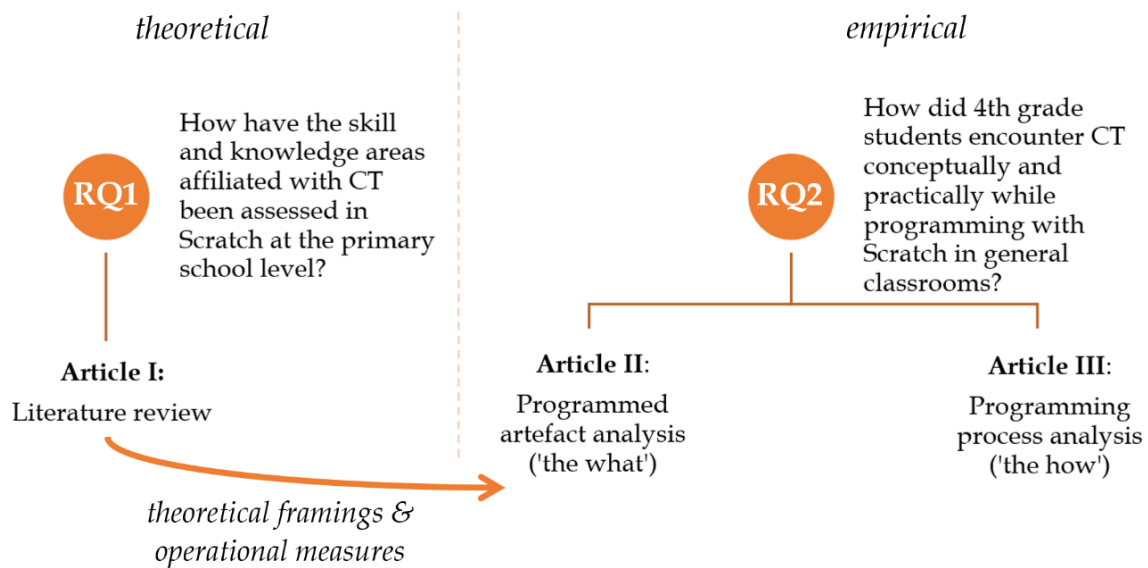


Figure 12. The mixed methods approach of analysing the data for the two RQs

they implement ('the what', in Article II) and programming activities that they carry out ('the how', in Article III). Respectively, mixed methods by ways of artefact analysis and a programming process analysis were found as a suitable pathway for combining these complementary assessments based on previous similar empirical studies (see subsection 2.2.5).

The main interest of this study is attaining rich empirical evidence of students' CT learning in general classrooms. Besides this interest, the analysis of CT-fostering programming contents and activities in RQ2 was accompanied by the previously established additional interest of potentially enacting important formative assessment processes during students' hands-on learning processes. The analyses of the data thus aimed at focusing on introductory programming contents and activities that could be potentially meaningful learning goals for students, valid indications of learning, and appropriate targets of feedback. In Article II, a content analysis framework for creative, introductory-level programmed features in Scratch projects, such as animations and user interaction, was found as a meaningful pathway to combine these interests. Article III, in turn, focused on the more social and process-oriented or temporal aspects of programming, which manifested expectedly much less visibly in static programmed artefacts.

The more detailed measures of analysing the data in the articles are described under the articles' summaries (in chapter 5) and in the original articles.

## 4.5 Research philosophy

Epistemologically, the literature review (in Article I) was based on rationalistic principles, that is, theoretical reasoning, or in other words the structured presentation of existing knowledge, to discover new truths (Schryen et al., 2015). In turn,

the case study (in Articles II and III) is based on the paradigm of empiricism, which is characteristic of educational sciences and emphasises the value and reliability of making observations of reality and drawing conclusions from them (Given, 2008). The empirical study is accompanied by a pragmatic interest, an attempt to produce practically applicable contributions to the topic (Yin, 2012). This interest can be summarised as the attainment of pedagogical knowledge for educational practice, particularly the more successful inclusion of CT and programming in primary schools. Moreover, this study examines learning, that is, a complex real-life phenomenon, and is founded on theoretical underpinnings construed by the author from a maturing research topic to discern real-life occurrences. The study can therefore be said to have an interpretivistic rather than a purely objective and value-free position (Mack, 2010).

The case study part of this thesis adopts a mix of exploratory and descriptive aspects (Yin, 2012) as it seeks to discover new knowledge and explain its characteristics richly in the current ill-understood state of research on this topic. Data collection in authentic classrooms was considered particularly important for these purposes. Authentic classroom situations were expected to highlight important but little studied and therefore inadequately understood pedagogical issues in CT. These included CT learning in time-constrained classroom settings, variety among the learning experiences of different kinds of students in compulsory education and the quality of instruction and guidance regarding the novel subject matter in non-controlled classrooms. The expectation was that the empirical findings could be reflected in tandem with theories in social constructivism, constructionism and creative computing education to produce applicable pedagogical models for classroom contexts. The findings were primarily intended for further comparison (e.g. affirmation, correction, specification) in other practical contexts and the further creation or specification of theoretical models and hypotheses for formal experimentation (Yin, 2012).

The intended explorations and descriptions in the case study prompted the utilisation of both quantitative and qualitative (i.e. mixed) methods, thus leading the study to abide by both holistic and atomistic evaluations of the acquired evidence. More concretely, a pivotal question is whether the investigation of CT can be approached through small and separable concepts or whether the phenomenon is something else other than the sum of its parts. Although both notions are employed while interpreting the results, the interpretations put more emphasis on the qualitative dimension. In other words, although the analyses utilised important knowledge gained by examining quantities, the study does not systematically perform, for instance, statistical tests. Instead, it draws on the foremost strength of qualitative methods: the in-depth study of a complex issue.

## 4.6 Ethical aspects

This study followed the ethical code of conduct guidelines provided by the University of Jyväskylä, the organisation where this research was conducted, and

The Finnish National Board on Research Integrity<sup>18</sup> (TENK) throughout the entire research process. Prior to data collection, the plan for data collection was presented to the principal of the participating school, the participating teachers and students and the legal guardians of the participating students in a written letter. The letter also included informed consent (Appendix A). The regular teachers of the classes gave the letters to the legal guardians in a parent-teacher meeting so the guardians could familiarise themselves with the study information without urgency at home. A signed permission slip marked with either 'participate' or 'not participate' was returned to the school by the students. An opportunity to cancel participation at any point of the study was offered to the principal, the teachers, the students and the students' legal guardians.

To deal with the students' concerns about the data collection instruments (e.g. the GoPro cameras), the instruments were brought to the school prior to the data collection and the students given an opportunity to see how they worked. The cameras were set in place for a mock session without recording. Additionally, the investigator explicitly told the students that the capturing devices were not intended to evaluate their behaviours but to merely record naturally occurring events.

A data management plan was devised and upheld based on the university's adapted guidelines complying with current legislation on data protection (e.g. GDPR<sup>19</sup>). In particular, to protect the privacy of the participants, all data was anonymised after data collection (e.g. the names of the students in Article III are all pseudonyms). All raw data was stored safely on an encrypted hard disk behind locked doors and in password-protected cloud storage.

---

<sup>18</sup> <https://tenk.fi/fi/tiedevilppi/hyva-tieteellinen-kaytanta-htk>

<sup>19</sup> General Data Protection Regulation: [https://europa.eu/youreurope/business/dealing-with-customers/data-protection/data-protection-gdpr/index\\_en.htm](https://europa.eu/youreurope/business/dealing-with-customers/data-protection/data-protection-gdpr/index_en.htm)

## 5 SUMMARY OF ARTICLES

This section summarises the aims, analyses, main results and relevant points of discussion in the three articles establishing the broader efforts of this thesis. An overview of the RQs and research designs, including the datasets used, of each article are presented in Table 2.

Table 2. Overview of the articles

Article	Aim	Design
I	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	A systematic literature review of 30 identified previous studies
II	Gain rich empirical insight of 4 <sup>th</sup> grade students' CT by assessing Scratch projects that they designed during a programming course	Empirical analysis of Scratch projects (N=325) made by 4 <sup>th</sup> grade students (N=57)
III	Gain rich empirical insight into 4 <sup>th</sup> grade students' CT by analysing the programming activities they carried out in pairs (dyads) while programming creative projects as final assignments in the programming course	Empirical analysis of the project plans and programming processes (video data) of 4 <sup>th</sup> grade student dyads (N=4)

### 5.1 Article I: Computational thinking in programming with Scratch in primary schools: A systematic review

#### 5.1.1 Aims

This article aimed to contextualise CT comprehensively in the Scratch programming environment for teaching and learning in primary school classrooms and

to explore the assessment of CT through Scratch in this context. In practice, the article included a literature review on studies involving assessments of Scratch programming contents and activities at the primary school level (K-9). The objective of the review was to gather Scratch programming contents and activities, use the defined theory of CT (i.e. the CEPs, see sub-section 2.1.4) 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. The RQs were:

- 1) What Scratch programming contents and activities have been assessed in K-9?
- 2) How have Scratch programming contents and activities been assessed?
- 3) How do different Scratch programming contents and activities contextualise CT concepts and practices via the core educational principles?

### 5.1.2 Methods

The article employed a systematic literature review method. Extensive literature searches were performed to find peer-reviewed studies focusing on the assessment of Scratch programming contents and activities at the K-9 educational level. After defining and employing exact inclusion and exclusion criteria for 432 search results in select journal databases (e.g. Association for Computing Machinery, ScienceDirect) plus additional searches on Google Scholar and 'snowballing', 30 publications were selected for review.

The Scratch programming contents and activities assessed in the publications were described based on their type (RQ1) 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 CT's core educational principles (in subsection 2.1.4, also listed in the original article) that they contextualised (RQ3). The analysis resulted in rubrics for Scratch contents and activities that can be rationalised to foster students' skills and knowledge in cross-contextual CT. Furthermore, the discovered methods for assessing CT based on the contents and activities were examined according to how they potentially enabled formative assessment processes as presented by Black and Wiliam (2009).

### 5.1.3 Main results

Prior studies focusing on programming in Scratch in K-9 involved the assessment of various kinds of programming contents and activities with diverse assessment methods and taxonomies or rubrics. Four distinct programming substance categories were found and called 'code constructs', 'coding patterns', 'programming activities' and 'other programming contents' (illustrative examples are demonstrated below). Methods of analysing the said contents and activities included, for example, artefact analyses, teacher observations and interviewing. The assessment taxonomies or rubrics included, for example, the presence or frequency of particular contents in programmed Scratch projects, the



description of students' behaviours and the completion percentage of pre-designed contents in projects.

Only six studies considered the direct assessment of cross-contextual CT through the contents and activities, and the remaining studies assessed them with or without presenting CT as a motivational theme in the study. Nevertheless, the contents and activities were found to contextualise the core educational principles of CT in manifold ways regardless of prior studies having directly established similar links. However, particular CEPs were not necessarily very straightforward to contextualise in Scratch discretely or in sufficient depth to assess them meaningfully (e.g. 'out-of-the box thinking', 'making decisions and reaching conclusions', 'finding and collecting data', 'identifying real-world applications of CT').

An example of an assessment instrument that analysed CT through programmed code constructs in Scratch projects was a web-based automatic analysis tool called Dr. Scratch. Students can use Dr. Scratch by inserting the web address of their Scratch project into the tool. The tool then evaluates the presence of specific coding blocks in the project as an indication as to what competence the designer has regarding particular key concepts in CT (e.g. an 'if' block in the project code stands as evidence for competence in Logic) (Moreno-León et al., 2015). Several other studies assessed the use of code constructs in similar ways without directly aligning them with CT. Nonetheless, several studies were found to contextualise or concretise CT in different ways. For instance, the 'initialisation' code construct sets initial values for sprites' properties, such as their size or position on the computer screen. This construct contextualises the notion of designing sets of instructions that start from an initial state (a fundamental CEP in the concept of Algorithms).

An illustrative set of examples of assessing CT through 'coding patterns' (i.e. code constructs combined in semantically meaningful ways to establish a larger computational structure) was found in a publication by Seiter and Foreman (2013). The authors reported using an analysis model with which they observed and categorised the presence and progression level of coding patterns, such as 'Animate looks', 'Maintain score' and 'User interaction', in students' game, animation and storytelling themed Scratch projects. The presence of specific patterns in a project indicated an estimation of the student's proficiency levels in particular areas in cross-contextual CT.

Ways of assessing 'other programming contents' that were found to contextualise CT were discovered in several studies. Previous studies examined, for instance, the number of different scripts and different sprites in students' projects (contextualising particular CEPs in Abstraction and Problem decomposition) and the meaningful naming of data variables (in Abstraction and Modelling and design, respectively).

As examples of assessing students' CT through their 'programming activities', Burke (2012) described and Ke (2013) examined the amount of students' effectuation of particular design phases (e.g. drafting, analysing scripts and testing)

while designing their projects. Such phases contextualised ways in which students can carry out the cyclical design of projects (a key CEP in the practice of Iteration).

#### 5.1.4 Discussion

The CEPs that were not very straightforward to contextualise in Scratch can be too vague or general to be taught distinctly in programming. They can also manifest more meaningfully when using other programming environments and environments that can promote engaging learning activities for novice programmers (e.g. Lego Mindstorms, the App Inventor) in compulsory education.

In terms of the CEPs that were rather straightforward to contextualise in Scratch, this article provided two new operational rubrics. The first rubric produced by this review is that of 'CT-fostering Scratch programming contents' (the 'what'). In short, students' understanding and skills in CT are fostered by implementing and can be assessed based on code constructs (e.g. 'loop', 'variable'), coding patterns (e.g. 'change location') and other programming contents (e.g. sprite naming) in Scratch projects. The rubric prompted a novel analysis of a more conceptual view of cross-contextual CT from students' Scratch projects as was carried out in Article II.

The second rubric produced by this review, modelled in adjunction to the notion of CT as a problem-solving process, includes the 'CT-fostering programming activities' (the 'how'), such as collaboration, iterative design, and testing and debugging, in and around programming with Scratch. The activities depict the computational problem-solving practices that students can develop and employ when designing Scratch projects. The activities may leave traceable evidence in projects as static content (e.g. remixed contents) but may be more thoroughly typified in students' programming processes. The rubrics prompted a novel analysis of a more practical view of cross-contextual CT from students' Scratch programming processes as was carried out in Article III.

The summaries should not be regarded as complete, because CT is a developing body of holistic skills and understanding in computational problem-solving (Tedre & Denning, 2016; Wing, 2006). The view of CT adopted here is relatively inclusive, and it can encompass areas that can be positioned in the more 'central' or 'peripheral' zones of CT that can be included or excluded as needed. Additionally, due to their versatility, simple descriptions of the contents and activities were considered a reasonable starting point. Thus, the contents and activities could be interpreted to contextualise different areas in CT in various ways.

Some of the summarised CT-fostering contents and activities could be meaningfully examined using objective metrics (e.g. presence of code segments), whereas others may be more subjective in nature (e.g. creative expression). On a related note, 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 additionally processing learning contents in other curricular areas (Garneli et al., 2016; Moreno-

León et al., 2017). 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 (Brennan & Resnick, 2012). Meaningful learning thereby includes authentic problems and selections of projects. In terms of CT in such settings, it is important to particularly focus on how students are thinking as they are programming (Lye & Koh, 2014).

In terms of formative assessment, first, holistic assessment should recognise the diversity of problem-solving situations and align contextualised, task-specific assessment rubrics to the focal areas of CT (Grover et al., 2017; Moreno-León et al., 2017). Educators can use precise CT-fostering rubrics (e.g. coding patterns and their underlying code constructs) as indirect CT learning goals and criteria.

Second, as programming is a demonstration of CT (Grover & Pea, 2013), the contents that students implement in Scratch projects can be viewed as evidence of their CT. Although the examination of code constructs within semantically meaningful coding patterns improves the validity of the assessment, programming projects are not direct measurements of thinking (Seiter & Foreman, 2013). It is crucial to complement the assessment by examining programming processes (Grover et al., 2017). According to the review, prior studies assessed students' programming activities via, for instance, observation, discourse analysis and interviewing. In schools, complicated research-designated tools are time-consuming. Additionally, prior studies assessed only certain CEPs and not CT comprehensively. Therefore, project content implementation could be examined alongside both peer-to-peer (see e.g. Israel et al., 2016) and student-project (see e.g. Ke, 2014) interactions (see Articles II and III).

Third, 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 (Lye & Koh, 2014). New methods for enabling such processes could be developed (e.g. micro-programmatic analysis of instantiated coding patterns) (see also Vihavainen et al., 2013), and existing automated assessment tools (e.g. Moreno-León et al., 2015) that cover some areas of CT could be revisited to better satisfy this need.

## **5.2 Article II: Assessing 4<sup>th</sup> grade students' computational thinking through Scratch programming projects**

### **5.2.1 Aims**

This article aimed to acquire rich empirical insight regarding 4<sup>th</sup> grade students' CT by assessing Scratch projects they designed during a programming course (artefact analysis). It revised a profound assessment framework and used it empirically to assess the programming contents and indicative cross-contextual CT in a sample of primary school students' (N=57) Scratch projects (N=325) (see description of data in section 4.3). The objective of the assessment was to investigate

students' conceptual encounters with CT in a comparatively comprehensive and fine-grained manner in naturalistic classroom situations and to evaluate the significance of the obtained evidence in CT education and the next steps in developing formative assessment of CT in schools. The RQs were:

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

### 5.2.2 Methods

The priorities in the analysis of the students' projects were to aggregate manifold programming contents indicating CT in Scratch thoroughly and systematically, to analyse programmed contents in the projects primarily through coding patterns (in particular, individually instantiated patterns and their underlying code constructs) and to account for the context of the contents rather than merely their technical presence.

An assessment framework was revised from various prior works (e.g. Meerbaum-Salant et al., 2013; Moreno-León et al., 2015; Seiter & Foreman, 2013). It was designed to assess three kinds of meaningful programming contents – individually instantiated coding patterns (e.g. 'Animation', 'User Interaction'), the types of those instances (e.g. 'Timed animation', 'Keyboard input') and the code constructs (e.g. 'control', 'coordination') that establish those particular instances<sup>20</sup>. This 'instantiated coding patterns-first' analysis was intended for semantic meaningfulness in creative design and for the possibility of micro-programmatic assessment (e.g. for targeted feedback). Based on the literature review carried out in Article I, the students' conceptual encounters with the core educational principles in CT's concepts and practices were logged for each student through all the implemented contents in their projects.

The method enabled the examination of the qualities and quantities of the students' conceptual encounters with specific CEPs in the following CT concepts/practices: Abstraction, Algorithms, Automation, Coordination, Creativity, Data, Logic, Modelling and design, Patterns and Problem decomposition.

### 5.2.3 Main results

The comparatively inclusive view of what students can learn about CT through Scratch and the use of a profound assessment framework resulted in ample and manifold empirical findings of contents programmed by the students and respective indications of their conceptual encounters with cross-contextual CT at different times of the programming course. In particular, the results yielded an assortment of detailed insight about the versatile ways in which the students implemented programming contents in the 'tutorial', 'remix', 'debugging' and 'design'

---

<sup>20</sup> See detailed rubrics in [https://www.researchgate.net/publication/344411984\\_Assessing\\_4th\\_Grade\\_Students'\\_Computational\\_Thinking\\_through\\_Scratch\\_Programming\\_Projects](https://www.researchgate.net/publication/344411984_Assessing_4th_Grade_Students'_Computational_Thinking_through_Scratch_Programming_Projects)

projects they programmed during the course. This insight is exemplified here selectively and partially through three viewpoints: chronological, programming content, and the students' project portfolios.

From a chronological viewpoint, the students' first projects (P1) – more discovery-focused experimentations with different features in Scratch – displayed evidence against spontaneous learning of event-driven design of algorithms. As much as 58% of all coding pattern instances in all P1 projects were dysfunctional. Programmatically, this was caused by the missing 'control' and 'coordination' constructs in the scripts. Although the presence of these constructs increased greatly in subsequent projects (see below), they were still occasionally missing throughout the course, indicating recurring difficulties with these basic features or projects that were not finished in the given time. 'Initialisation' remained a frequently missing construct throughout the course.

The median and mode completion rates of the highly structured tutorials (P2) made after the discovery-oriented first session were only 50% (four of eight coding pattern instances). However, these projects did entail contents that were not covered by the tutorial, indicating that the tutorials were left incomplete prior to proceeding to creative design or that contents were modified after completing the tutorial. After completing the tutorial, the presence of dysfunctional coding pattern instances decreased to a mere 12% in the second design projects (P3) (and remained at a similar level throughout the programming course).

From a programming content viewpoint, two separately introduced collections of debugging challenges (P4 and P7) each comprised four faulty projects that the students were guided to begin correcting, potentially submitting all four of them. The number of total submissions in each collection and the correctness of those projects both varied. In particular, the number of submitted projects decreased drastically in the face of a project involving faulty 'looped animation of location' with 'move' and 'bounce' blocks. In turn, 95% of the students correctly debugged a 'repeat until' block, which was not introduced during the course prior to the challenge.

In projects that had pre-set programming objectives, the students implemented the contents that were minimally required of them correctly for the most part. Among these projects was remixing an incomplete project (P5) and implementing an instance type called 'Event-sync animation (location)' with the 'initialisation' code construct for two separate sprites in it. Other mainly correctly implemented pre-set programming objectives also entailed creative games (e.g. P6) that asked questions, received keyboard inputs as responses (i.e. the 'Keyboard input' instance type) and evaluated the correctness of the answers (i.e. a 'Data manipulation' coding pattern involving the testing of a stored variable). Few students failed to meet the pre-set objectives due to unscripted blocks or the lack of particular key code constructs (e.g. 'initialisation').

In projects where the students had more creative freedom (P1, P3 and P8), 'Animation' was by far 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. Similarly,

volitionally designed coding patterns in all the projects made during the course were by far mostly 'Animation', 'Speech and sound', and 'User interaction'.

Ultimately, the students' portfolios, which were aggregated by the investigator from all projects each student had submitted, contained between 2 to 14 projects (Median = 10), indicating that few students participated in designing only a few projects or that all portfolios did not include all projects programmed during the course. Nevertheless, the portfolios demonstrated the varying number of instance types of coding patterns the students programmed during the course. Among the most common types were 'Event-sync animation', 'Monologue' and 'Green flag'. More than half of the solution types had 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; nonetheless, among the most common constructs were 'sequence' (*Mdn* = 55), 'repeat'/'forever' (*Mdn* = 22) and 'green flag' (*Mdn* = 16).

According to the students' conceptual encounters in CT, as indicated by the programming contents in their project portfolios, all the students re-instantiated the coding patterns and code constructs (Patterns) and decomposed the projects into smaller parts (Problem decomposition). Nearly all (>90%) the students designed complex projects (Abstraction), implemented algorithm control structures and 'initialisation' (Algorithms), remixed (Collaboration) and utilised logical operators (Logic). However, less than half (<50%) of the students abstracted behaviours for sprites (Abstraction), used procedures (Algorithms), utilised I/O devices (Automation) and synchronised parallel scripts (Coordination). None of the students implemented recursive solutions (Algorithms) or Boolean logic (Logic). Variation was large among instantiating synchronised parallel scripts (Coordination), demonstrating that the few students who encountered this principle did so several times.

#### 5.2.4 Discussion

Although the results confirmed findings in previous studies, the unique framework allowed revealing novel insight. Most importantly, assessing programmed contents 'instantiated coding patterns-first' allowed attaining insight regarding students' implementation of CT-fostering programming contents in diverse creative circumstances that were semantically meaningful, thus also favouring the legitimacy of the examination.

Among the expected results was that the more prevalent conceptual encounters could be expected to occur somewhat naturalistically in Scratch. At the programmatic level, the more common conceptual encounters were very likely caused by the instantiation of particular contents, namely those that are integral to designing interactive media (e.g. the 'Animation' coding pattern) (Brennan & Resnick, 2012), those that are integral to the nature of event-driven programming in Scratch (e.g. the 'event-sync animation' instance type) (Maloney et al., 2010) or those that are among code constructs or code blocks that novice programmers typically first learn to use (e.g. 'green flag') (Grover et al., 2014).

Perhaps most intriguing and relevant for pedagogical consideration was examining what the students rarely or never programmed (e.g. the 'Data manipulation' and 'Collision' coding patterns, instance types with conditional logic and state-sync methods, the 'repeat until' and 'custom variable' code constructs) and how this affected their conceptual encounters with CT. After all, several of the less-encountered CEPs even during the 12 programming sessions can be considered as fairly fundamental in terms of learning CT more exhaustively (Grover & Pea, 2018).

A significant underlying reason for the seldom encountered CEPs may come down to the types of projects designed. Moreno-León et al. (2017) showed that the presence of certain constructs typically varies between projects in different genres. The projects programmed by the students typically lacked usability and consequently resembled projects more for viewing than playing. Therefore, the students may have lacked opportunities to explore supplementary Scratch project genres, such as simulations or more sophisticated games for which the more rarely encountered contents may be more typical. Many of the programming contents were also not systematically introduced during the course, suggesting that students may be inclined to intentionally implement contents with which they are familiar. Furthermore, different kinds of external devices, such as microphones or extensions that Scratch inherently supports, which could have enabled designing more versatile projects, were not available in the school.

Examination of the contents programmed by the students in different kinds of projects at different times also suggested, above all else, that the students did not seem to intrinsically grasp controlling algorithms (Algorithms) and coordinating them with, for instance, timing (Coordination) at the beginning of the course. It was important to note that 'control' and 'coordination' became significantly more prevalent after the students had completed the scripting tutorial. On a related note, most students debugged the previously unused 'repeat until' block correctly in a structured debugging challenge (P7.4). This finding reinforced the 'use-modify-create'-like idea (see Lee et al., 2011) that debugging challenges (or remixing in general) could offer a viable route between direct instruction and more open-ended design to teach students to understand and use even more advanced constructs.

Examining contents implemented at different points in time also suggested that a conceptual encounter may not self-evidently guarantee gaining a deep understanding. This was primarily suggested by the notorious 'initialisation' code construct (see also Franklin et al., 2013; 2017), which was occasionally still missing towards the end of the course despite it having been directly instructed in P5. However, because the fairly fundamental 'control' and 'coordination' code constructs were also occasionally missing even in the final projects, suspicion arises as to whether all projects were fully completed in the allocated time. These inferences most importantly highlight the importance of complementing the analysis of students' CT by examining their programming processes, leading to Article III.

### 5.3 Article III: Fourth grade students' computational thinking in pair programming with Scratch: A holistic case analysis

#### 5.3.1 Aims

This article aimed to gain rich empirical insight regarding 4<sup>th</sup> grade students' CT by analysing the programming activities they carried out in pairs (dyads) while programming creative projects as final assignments in the programming course described in section 4.3. The article developed new pair programming process analysis methods and adopted methods from proximal research paradigms to explore and describe four student dyads' (D1, D2, D3, D4) initial project plans, programming processes over two programming sessions (Mean duration = 38 min, 0 sec) and finished Scratch projects. The objective of the analyses was to investigate interlinked programmatic and non-programmatic factors shaping students' Scratch programming processes and potentially regulating their shared CT learning, artefact-related outcomes, and non-cognitive aspects, such as their enjoyment in programming, in classrooms. The RQs were: How did the 4<sup>th</sup> grade dyads:

- 1) plan their open-ended Scratch projects? (Planning)
- 2) cyclically design the projects? (Iteration)
- 3) mutually participate in the design, activate teachers and peers, and search for external materials? (Collaboration)
- 4) locate and fix bugs? (Debugging)

#### 5.3.2 Methods

The studied CT dimensions targeted by the four above RQs were not directly identifiable in the data. Instead, smaller, more straightforwardly identifiable occurrences jointly denoting them in the data were analysed as follows.

The data was analysed in two main ways. First, by employing content analysis, the students' initial project plans and final projects were examined in terms what the dyads aimed to design (programming contents, graphics, audio) based on their initial project plans and how they eventually succeeded in doing so. The analyses utilised a framework for programming contents in Scratch projects adopted from Article II.

Second, the video data portraying the dyads' shared programming processes was analysed systematically in three overlapping layers – what happened on the computer screen (*design events*), who used the computer (*computer control*) and what kind of talk occurred (*talk*). For example, for the design events, the analysis pinpointed such events as 'inactivity', 'graphical design', 'coding' and 'testing play'. In turn, the analysis of talk differentiated 'silence', 'teacher talk', 'mutual talk' and 'peer talk'. These occurrences were further specified as, for example, different forms of 'help-seeking' or, in terms of mutual talk, 'disputing', 'telling', 'opening' and 'negotiating'.



The multilayered coding of the three layers produced a myriad of information, and main targets for interpreting relevant findings from the data were specified to answer the RQs. For example, for examining ways in which the students carried out 'testing and debugging', analysis focused mainly in and around 'bug reveal' occurrences in the design events layer. The interpretations utilised qualitative and quantitative methods: for instance, when examining how the students carried out 'iteration', the temporal positions of different design events during the sessions were examined as a whole.

The analysis method enabled the examination of the quality and quantities of the students' shared practical encounters with specific CEPs in the following CT concepts/practices (referred to in the article as 'CT activities'): Project planning and modelling, Iterative design, Collaboration and Testing and debugging.

### 5.3.3 Main results

The four dyads had planned assorted creative projects to be programmed over the two final sessions of the programming course. The data revealed several ways in which the dyads carried out the CT activities while the versatile creative plans materialised into the projects (described in more detail in the original article). The main results are exemplified below more in terms of their generic meaning rather than the intricate analysis and coding of the data.

Dyad 1, involving students Sami ( $S_{17}$ ) and Pete ( $S_2$ ) from class  $C_a$ , had planned a tower defence game. Their plan included pseudo-code-like depictions of both basic level features (e.g. motion animations) and advanced ones (e.g. a timer). Sami had designed a part of the project at home prior to the first design session. During the two sessions, the students tested play after coding and coded after having revealed bugs much more often than effectuating other design events. They also altogether effectuated approximately five times more productive than unproductive design, such as actual coding. All unproductive design was rather insignificant except for a sequence of frustrating mishaps when failing to use the graphical editing tools. However, despite the seemingly efficient design, the students had exceptionally imbalanced roles: computer control was largely one-sided (Sami: 78% driving, Pete: <1%) and Sami was approximately five times more verbal in one-sided talk. Nonetheless, Pete once complemented Sami's skills in debugging by pointing out a bug and participating in the negotiation while finding solutions. The students' debugging once also revealed an ineffective approach: modifying irrelevant code in place of code that would have fixed the bug. Furthermore, in another bug, the regular teacher of the class ( $T_a$ ) was unable to help the students. In turn, the visiting teacher explicated the solution, but the students did not seem to grasp the solution by facing a similar bug again later. After the two design sessions, D1's final project was lacking the more advanced features.

Dyad 2, involving Johanna ( $S_4$ ) and Mari ( $S_8$ ) from class  $C_a$ , had planned a Harry Potter themed story. Their plan included pseudo-code-like depictions of exclusively basic level contents, such as animations and user interaction, next to descriptions of backdrops and sprites for the project. The students began with a

rather extensive focus (approx. 21 minutes) on graphical design before continuing to coding. D2 showed similar effective iterative tendencies concerning testing play and coding to D1 with the addition that no unproductive coding transpired. However, D2 transitioned as often as fifteen times to unproductive graphical design, mainly involving discouraging difficulties with drawing sprites in the graphical editor. Moreover, participation within D2 was also highly irregular based on amount of driving (Johanna: 78%, Mari: 10%) and one-sided talk (Johanna: 70%). Johanna also showed dominative behaviours while, for instance, exhibiting unilateral ineffective coding by merely guessing what she should do while programming. Although the students switched roles, disputing and computer conflicts occurred while Mari appeared to want to smuggle in single-minded ideas when given the rare chance to drive. The students also encountered altogether ten bugs, four of which appeared substantial for involving the same construct: initialisation. D2's final project comprised a majority of the features they had initially planned.

Dyad 3, involving Marja (S<sub>18</sub>) and Anne (S<sub>29</sub>) from class C<sub>b</sub>, had planned a princess rescue game. Their plan included both basic and advanced level features planned in generic human language and occasional pseudo-code resembling Scratch blocks. The students' first session was dedicated predominantly to designing the graphics, and their design was altogether efficient in ways similar to D1 and D2. Despite presenting their project to a peer at the end of the first session, there was no substantial project-related talk with peers among any dyad. Also similarly to the previous dyads, participation was uneven in terms of driving (Marja: 77%, Anne: 11%) despite high amounts of negotiation, harmonious switches in driving, and help-seeking from the partner. The most substantial findings concerned the high amount of inactivity (44% of the time), which transpired with interlaced bug revealing (84% of all playtests). In particular, the students were unable to implement their plans for the more advanced contents by, for example, adjusting irrelevant scripts and receiving variably effective help from the regular teacher (T<sub>b</sub>). Ultimately, D3's final project was relatively small for lacking the more advanced features.

Dyad 4, involving Tinja (S<sub>52</sub>) and Saana (S<sub>54</sub>) from class C<sub>c</sub>, had planned a beach story. Their plan encompassed human-language descriptions of graphics and a narrative progressing through exclusively basic level features, such as animations. Their rather short focus on graphical design allowed them to effectuate altogether mostly coding (27% of the time) while altogether designing efficiently based on similar tendencies for productive design and testing play often as discussed with the previous dyads. Despite the high amount of negotiation (66% of all mutual talk), computer control was again highly unbalanced (Tinja: 95%, Mari: 2%). Tinja even chased Saana away from the computer thrice, increasing her personal account of one-sided talk and especially the amount of 'telling'. Moreover, the dyad demonstrated bug workarounds in place of programmatic fixes, inability to generalise the abstractions of previously implemented contents, and inefficient coding in form of mere guessing. D4's final project encompassed more than what they had planned.

### 5.3.4 Discussion

Few previous studies have focused on students' CT-fostering programming activities in pair programming in Scratch in primary school classrooms. Geared with a combined exploratory-descriptive design, rich data and a multilayered analytical methodology, this case study specified results regarding students' Scratch programming gained in previous studies (e.g. Burke, 2012; Ke, 2013; Lewis, 2011; Shah et al., 2014; Tsan et al., 2018), provided novel empirical evidence for theories in programming education and especially in terms of CT and pointed the way for additional and more focused research.

The examination of the dyads' planning (Project planning and modelling) next to the subsequent design highlighted that plans fundamentally guided the design processes, stressing the importance of appropriate initial planning. More free-form planning (e.g. pseudo-code-like depiction, drawing; Burke, 2012) appeared characteristic to the students, but the translation of such plans into even previously implemented programming contents occasionally benefitted from various forms of instructional support (Carlborg et al., 2019; Lye & Koh, 2014). Altogether, initial plans implying unawareness of operable designs led to strenuous debugging, discouraging creative compromises, inefficient trial-and-error, and unfinished projects. On the other hand, free planning may guide students to merely implementing what they know and not what they could potentially learn. In fact, spontaneous planning and refinement through, for instance, browsing the graphics (Ke, 2013) and suggesting new ideas during programming (Campe et al., 2020) guided the design as well. Discovery as highlighted in constructionism (Brennan & Resnick, 2012) could be worthwhile, although it can demand intense support from knowledgeable teachers (Kong et al., 2020), which can be inappropriate in classroom settings. Planning could thus involve guidelines orienting toward basic core programmable features in more story-like projects at the introductory stage of learning in open-ended programming and regard more ambitious features as extraneous learning opportunities (Mayer, 2004).

The students' iterative design processes (Iterative design) displayed a typical progression from graphical design to coding, both of which were altogether perhaps slightly pressurised and hindered in the limited time of two 45-minute sessions. A priority between the more computational (Denning & Tedre, 2019) and the more self-expressive (Brennan & Resnick, 2012) emphasis may be required in open-ended Scratch programming in classrooms, for example, by instructing students' to utilise merely premade graphics. Nonetheless, the novice programmer students' autonomous design raised expectations for an inherent occurrence of staying on-task, testing play often, attempting to fix bugs, and make mainly productive modifications. The more exact quality of design, however, varied, as seen through such examples as guessing and adjusting irrelevant code (i.e. trial-and-error) (Ben-Ari, 1998).

Examinations of the social domain in pair programming (Collaboration) specified how students' participation may occur, highlighting control of the expectedly shared design process as a key issue. Most crucially, the drivers were

apparently afforded with privilege (Lewis & Shah, 2015; Tsan et al., 2020; Zakaria et al., 2019) to dictate how to activate the navigator regardless of the quality of their contributions. This setup evidently led to missed opportunities to implement new contents (Deitrick et al., 2014), thus also potentially learning (Brennan & Resnick, 2012), dissociation from the design (Lewis & Shah, 2015), and altogether dissimilar knowledge acquisition, interest, and enjoyment. Curiously, no rationale emerged for the programming roles, leading to the prospect that a risk of participatory imbalance can reside in self-directed pair programming without deliberate pedagogical consideration. For example, teacher mediation can be required in reflecting viewpoints and reconciling students' differences (Roschelle & Tesley, 1995; Tsan et al., 2021; Zakaria et al., 2019) in addition to mandatory role switches (Lewis & Shah, 2015) and distribution of design tasks. Furthermore, representing another aspect in collaboration, the lack of utilisation of external resources (Brennan & Resnick, 2012) showed that such practices may need to be explicitly facilitated.

Findings related to debugging (Testing and debugging) showed how bugs appeared to diagnose the students' programming incapacities (Ben-Ari, 1998) and, as a pedagogically desirable effect, lead learning to new places. The findings also specified where students' misconceptions can potentially reside at the introductory level in Scratch and how high loads of information (Lye & Koh, 2014) and the inability to generalise computational abstractions regularly (Brennan & Resnick, 2012; Grover & Pea, 2018) may hinder open-ended design, both appearing central for instructional support. Altogether, although learning through discovery and bugs appeared viable in this educational context, the quality of debugging was crucial. It depended on the students' individual capabilities and actions, such as employed help-seeking methods (Mäkitalo et al., 2011). Importantly, such ineffectual debugging practices as developing workarounds, adjusting irrelevant code, and guessing emerged, again stressing the importance of teacher knowledge and support (Kong et al., 2020) especially of effective programming contents and good programming strategies as were overviewed in Article I.

## 6 CONCLUSIONS

This study specified the educational goal of introducing CT through programming at the primary school level and evaluated ways to assess students' learning in terms of that goal in Scratch. It also developed new methods for assessing primary school students' CT richly and holistically in authentic programming situations and provided rich empirical insight about students' CT in the context of programming with Scratch. These actions were divided into three peer-reviewed scientific journal articles. This chapter begins by summarising the key findings of the three articles vis-à-vis the two formulated RQs (in section 6.1). The overall contributions of the thesis are subsequently discussed on a more general level (in section 6.2), the limitations of the study are elaborated (in section 6.3) and suggestions for further research and practical development are highlighted (in section 6.4). Last, the author makes closing remarks regarding the current developments, trends, challenges and prospects in this particularly heated educational topic based on this research (in section 6.5).

### 6.1 Summary of key findings

The first RQ of this thesis was 'How have the skill and knowledge areas affiliated with multifaceted CT been assessed in Scratch at the primary school level?' Article I (the literature review) showed that Scratch contextualises the different conceptual and practical areas in the adopted comparably inclusive view of CT as viewed through the CEPs in multiple ways. Particular CEPs may not manifest in Scratch very discretely, as they can be either too generic or can be better learnt in other programming environments. Otherwise, the practical affordances in and around Scratch can be understood to enable the manipulation of specific CT-fostering programming contents (code constructs, coding patterns, other programming contents) and the effectuation of specific CT-fostering programming activities. The contents and activities can be

assessed in several ways and can be used for instrumentalising processes in formative assessment—clarifying learning goals indirectly for CT through programming contents and activities, evincing student understanding through their ability to put those contents and activities into practice and providing feedback about the ways in which they do so while programming in Scratch.

The second RQ was ‘How did 4<sup>th</sup> grade students encounter CT conceptually and practically while programming with Scratch in general classrooms?’ Article II revised a framework for analysing CT-fostering programming contents (types of instantiated coding patterns and their underlying code constructs) from the 4<sup>th</sup> grade students’ Scratch projects they programmed during a programming course. The analysis provided rich empirical insight regarding the students’ conceptual encounters with CT through semantically meaningful creative features in their projects. The results suggest that particular conceptual encounters occur more naturalistically in this educational context, specifically through particular coding patterns and code constructs that are typical in animation and story-like projects. In turn, other conceptual encounters can require deliberate instructional planning and pedagogical thought. Recommendations for more structured instruction rather than mere pure discovery-focused learning, reinforcing previous learning by re-implementing programming contents and progressing towards the implementation of more game-like projects emerged in particular.

Article III employed a multilayered analysis to analyse CT-fostering programming activities that 4<sup>th</sup> grade student dyads carried out when planning and programming their final open-ended creative projects in the course. The results denoting the students’ practical encounters with CT suggest that initial project planning is vital in terms of both avoiding impending design pitfalls and offering appropriate learning opportunities. Spontaneous planning and discovery can also lead the process, however, by demanding intense support from knowledgeable teachers. Although dyads’ autonomous programming can in many ways be profitable, their programming processes, including the capability to proceed autonomously and require help, can vary greatly. In particular, specific undesirable actions regarding implementing and debugging programmed contents and social knowledge construction both within the dyads and in terms of activating external instructional resources can emerge. The findings highlight crucial areas of learning and the necessity of deliberate guidance alongside support for self-directed problem solving.

## **6.2 Contributions of the study**

The contributions of this study can be viewed from two perspectives. First, this study defined a tangible educational objective for CT in the context of programming at the primary school level as a theoretical premise for RQ1. The objective was first formulated as a general description in subsection 2.1.2 and subsequently concretised as CT’s CEPs in the context of programming in subsection

2.1.4. In other words, the thesis formulated a tactile outline of what CT can mean for primary education, particularly through programming education. As this idea has not been entirely clear in prior literature, this theoretical groundwork can be viewed as a meaningful contribution to the field<sup>21</sup>.

The second perspective concerns the developments gained by answering the RQs via the articles. In terms of RQ1, the literature review in Article I illuminated ‘second-order programming concepts’ in the context of the especially popular Scratch programming environment and mapped them to the ‘first-order cognitive CT’ as viewed through the CEPs. The main contributions of the article were therefore the rubrics for Scratch programming contents and activities contextualised systematically in CT. Although the results were specific to Scratch, the results are discussed in more generic terms from a curricular viewpoint (in subsection 6.2.1).

In terms of RQ2, the empirical case studies in Articles II and III built on the theoretical preparatory work in Article I and focused on authentic, creative pair programming situations in naturalistic classroom situations by means of artefact analysis and programming process analysis. More precisely, the articles attained rich empirical evidence of the students’ conceptual and practical encounters with CT based on contents they programmed in their projects and the students’ ways of carrying out CT-fostering programming activities in pairs. The main contributions of the articles were research-based evidence for teaching and learning CT in Scratch and methods of assessing CT in Scratch in primary school classrooms. To discuss these contributions coherently, they are organised in the theoretical context of ‘assessment for learning’, in particular in the processes depicted in formative assessment (in subsection 6.2.2), which was set as the overall pedagogical interest of this thesis.

### 6.2.1 CT in the curriculum

This study exemplified how introductory CT could be concretised comprehensively via core educational principles (in subsection 2.1.4). This theoretical framework could be interpreted as a set of conceptual and practical overall learning criteria for CT interpreted as a problem-solving methodology. This contribution is important, because two interconnected matters in CT have remained somewhat obscure: its educational intent and its consequent conceptualisation for teaching and learning in primary schools. The obscurity has been further enhanced by mixing programming in the pot as the enacted curricular subject instead of CT (see Heintz et al., 2015; Mannila et al., 2014). It is therefore important to stress the multiple potential educational benefits that CT can bring (e.g. teaching coding skills, generic cognitive skills, computational problem-solving skills and computational literacy). It is also important to be aware of the different ways to concretise its core skills and knowledge and, in

---

<sup>21</sup> It is vital to appreciate that the term CT is still young and prone to further shaping. Amid ongoing discussions of what kinds of skills are more central and more peripheral in CT, this study cultivated a relatively inclusive view of CT to find meaningful grappling points between curricular areas in educational practice.

particular, highlight the distinction between the two key conceptual domains in which teaching and learning can be examined: first-order cognitive CT, such as the concept of Algorithms, and second-order programming concepts, such as particular code constructs in Scratch.

Despite the rather lacklustre direct presence of CT in school curricula, such activities as programming and computing have made inroads in schools as both compulsory and elective subjects and through separate or integrated roles in other topics (Balanskat, Engelhardt, & Ferrari, 2017; Bocconi, Chiocciariello, & Earp, 2018; Heintz et al., 2015; Mannila et al., 2014). Illustratively, in the Finnish core curriculum (Opetushallitus, 2014) (see subsections 4.1.1 through 4.1.3), programming is mentioned in the ICT transversal competence (T5), which is expected to penetrate all school subjects. It is also defined more explicitly in the subjects of math and crafts. Although this study did not perform detailed curriculum analyses, it is safe to say that CT does not appear very sweepingly in the Finnish curriculum. For instance, 'algorithmic thinking' as mentioned in the curriculum (Opetushallitus, 2014, p. 379) is not clearly defined. Moreover, as a concrete example, to receive mark '8' at the end of the 9<sup>th</sup> grade, it is sufficient for a Finnish student to have programmed simple computer programs. The empirical data of this study revealed that such an achievement can be highly trivial for the multifaceted CT. Critically put, does programming a Scratch sprite to perform a dance animation constitute deep understanding of computing as a foundation of problem solving and a societal phenomenon?

It is important to note, however, that the types of thinking or thought processes (or: concepts and practices) affiliated with CT (e.g. abstraction, problem decomposition) can be rather universal and altogether present in different parts of the curriculum without distinct mentions of CT. Nonetheless, the inclusion of CT as a clearly defined competence would likely promote its more methodical adoption in teaching and teacher training. CT possesses a unique disciplinary conceptual and practical background (Denning & Tedre, 2019), which may need to be taught and learnt with deliberate measures rather than expecting that it spreads routinely and strongly as a stowaway competence via other learning. In other words, claims of CT being a competence intrinsic to all teachers appear to have no strong evidence (Denning, 2017) and may disregard the unique nature of CT as a deliberate practice of the discipline of computing with ICT tools in different real-life situations (see Figure 5).

Altogether, such programming environments as Scratch may facilitate taking small 'doses' of CT, as also demonstrated in the programming course organised in this study. However, contemporary pedagogy promotes programming and CT as ways to shape learning methods and learning processes by combining subjects in multidisciplinary learning throughout the school journey (Lonka et al., 2018). As justified by the ubiquitous role of computing in the world (Denning & Tedre, 2019) and to advocate for computing in multidisciplinary learning (Lonka et al., 2018), CT and programming could transcend the curriculum much more concretely and expansively than in current curricular guidelines in Finland. This is also emphasised in part by the obtained evidence,



which suggests that single or even multiple conceptual and practical encounters with CT may not ensure very deep learning.

Simply put, CT will need sturdier forms in school curricula in the coming years. Despite CT having been introduced in the context of various school subjects in previous research (see subsection 2.2.1), a systematic integration of CT in the curriculum would require further effort. The interest in introducing CT in cross-disciplinary STEM or STEAM topics (e.g. Hutchins et al., 2018; Kafai et al., 2019; Pears et al., 2019) is a step ahead, but juxtaposing CT with STEAM overly strongly may add to the risk of CT and programming becoming more profiled as topics that are irrelevant in other school subjects. Studies have begun to integrate CT in a variety of subjects, including the humanities (see subsection 2.2.1), and bolstering such endeavours can be extremely fruitful for the more expansive incorporation of CT in schools.

Providing justification for its validity and breadth, the theorisation for CT in this study relied on several publications on CT from earlier cross-curricular frameworks (e.g. Settle & Perkovic, 2010) to more recent scholastic attempts to bring clarity to the competence (e.g. Grover & Pea, 2018). The purpose of the atomisation of CT into CEPs was particularly to clarify the components of the competence to attain a tangible framework for research and practice. It is still vital to be mindful of the more 'gestalt' function of CT in the design of meaningful computational artefacts rather than as a collection of manoeuvres that are exploited one by one or 'checkboxes' to mark as 'taught' in the curriculum (see also Voogt et al., 2015).

Despite the above, there is justification for the notion that some parts of CT (e.g. the design of algorithms) are taught in isolation in schools by, for instance, engaging students to program primitive robots (e.g. Bee-bots) to acquire fundamental computational concepts (Grover et al., 2019). Earlier school grades could orientate students towards using technology and understanding simpler computational models to become fluent in them. The more far-sighted educational objective in CT can be to deepen understanding in select ways by playfully computing solutions for real life-like problems in new and interesting learning situations. The versatile computational techniques, models and ideas depicted in the CEPs could thus be taught and learnt across the curriculum in situations where they are applicable.

The skills and knowledge displayed via the CEPs and contextualised more exactly in Scratch contribute to the whole competence of CT, but they should not be seen as all-encompassing. Scratch (or any other specific programming environment) is typically intended for a specific kind of computational aspiration and is thus likely to have limitations with respect to fostering CT exhaustively. CT-related skills, such as efficient design (see e.g. Csizmadia et al., 2015; Shute et al., 2017), complexity management (see e.g. Angeli et al., 2016; Hsu et al., 2018), 'out-of-the-box' thinking (see e.g. Grover & Pea, 2018) and making decisions and reaching conclusions (see e.g. Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018) are intricate cognitive tasks that likely manifest in

different ways, which can also be held as a criticism of their overly generic definitions. Despite all, no computational tool or environment likely facilitates the learning of CT all-embracingly or demonstrates how computing can be performed across a very broad selection of problem-solving situations. To account for both the versatility of CT and the plurality of students learning it, CT should be taught and learnt in various contexts, some of which can focus more or less on, for instance, research, the creation of artefacts or creativity (Nijenhuis-Voogt et al., 2020). By any established definition of the competence, such themes as robotics, digital fabrication, game-making, simulation design and web design are each very important CT-fostering topics to include in teaching and learning.

The breadth and non-hierarchical quality of the substance established in the CEPs can, however, pose a challenge for integrating CT and programming progressively into a curriculum and guiding the design of learning modules at the grass-roots level. Until the repertory increases in hierarchy, key questions may include: What are more central and more peripheral skill or knowledge areas that all students should learn? What is the appropriate criteria for mark '8' at the 9th grade when considering the multifaceted nature of CT? How should the different ways of putting CT into practice (i.e. programming and other ways) be translated as generic guidelines? More research to answer such questions is needed, although the results of this study were able to provide prefatory outlines, especially from the viewpoint of programming with Scratch (see subsection 6.2.2). Still, if the general purpose of primary education is fostering an understanding of the world and preparing for further studies and work life, then perhaps a taxonomical approach can serve as an initial guideline in setting the bar in CT education: 'a tool for everybody (basic-level understanding), a professional tool for some people (advanced-level problem solving)'.

Another crucial layer to consider in the discourse regarding CT in the curriculum is the computational literacy aspect in CT. The core question in this regard is how learning activities and activities beyond the planning and design of artefacts can contribute to students' expanding perspective on computing as a social and societal phenomenon (see also Bocconi et al., 2018; Høholt et al., 2021; Kafai et al., 2019; Lonka et al., 2018; Williamson, 2016). For instance, Mertala et al. (2020) argue that the structure of power generated by algorithms penetrating societal reality and controlling people's behaviours is not currently taken into account in primary education. Programming is presented through logical exercises that seem free from such values as participation, democracy and making a difference, which are otherwise emphasised in the Finnish curriculum and which are also relevant in the societal role of coding. A presented solution is examining coding as a textual event and code as a socio-material text that has societal and social consequences. Programming is not just a functional process or a skill needed for the workplace to satisfy economic growth. An extremely important but possibly enormous endeavour would be including CT and programming more resolutely in the context of multiliteracy – showing students gradually what kinds of possibilities and challenges accompany technology and algorithmic practices (Mertala et al., 2020). Embedding the dimension

of CT as problem solving *within* this approach could be a justified ‘big picture’ of CT in the curriculum.

## 6.2.2 Formative assessment of CT in Scratch

### 6.2.2.1 Clarifying learning goals

This study construed cross-contextual CT concretely through CEPs and examined how the CEPs can be contextualised as—and potentially learnt through—programming contents (‘the what’) and programming activities (‘the how’) in Scratch (second-order programming concepts). In contrast to the rather intangible portrayals of CT learning goals in earlier literature, the concrete CEPs can be used to clarify learning goals or criteria for learning directly for CT. This idea follows the notion that the starting point of any educational intervention is the educational goal—what students are (individually or collaboratively) expected to learn (Black & Wiliam, 2009). To understand how particular second-order programmatic affordances in different practical contexts (e.g. the code blocks in a programming environment) necessitate learning about the CEPs and/or foster their learning, the CEPs may require further contextualisation. In this spirit, the comprehensive rubrics for Scratch programming contents and programming activities presented in Articles II and III can be used to clarify indirect but contextualised cross-contextual CT learning goals for students in the context of Scratch.

The contents and activities can in themselves be meaningful targets of learning; they serve as computational models, ideas or techniques that aid in establishing solutions to different problems in the design of Scratch projects. Learning can be defined as understanding the contents and activities as conceptual bodies and practical approaches and implementing them in the practice of programming as appropriate contents and activities. The rubrics may be adapted in other graphical programming environments as well, but thorough rubrics would likely necessitate a systematic contextualisation.

In particular, the comprehensive categorisations for semantically meaningful CT-fostering programming contents—coding patterns and code constructs as represented as graphical code blocks and their combinations in Scratch (see examples in Figure 13 and detailed rubrics in Article II)—can operate as core computational ideas that are presented and explored in early and intermediate levels of learning programming in schools. The myriad contents shape meaningful creative features in the design of authentic animations, stories and games. Therefore, understanding and using them can be seen fundamental in the design of purposeful computational models for media projects in Scratch (Seiter & Foreman, 2013), potentially while processing the substance of other curricular areas (Moreno-León & Robles, 2016).

The process of assessment for learning incorporates a notion of systematising ways to guide the interpretations of assessment results and responses to those interpretations in a formative way. In other words, Black and Wiliam (1998) highlight the importance of ‘criterion sequences’ (also referred to as learning progressions, learning trajectories or learning paths) as models of

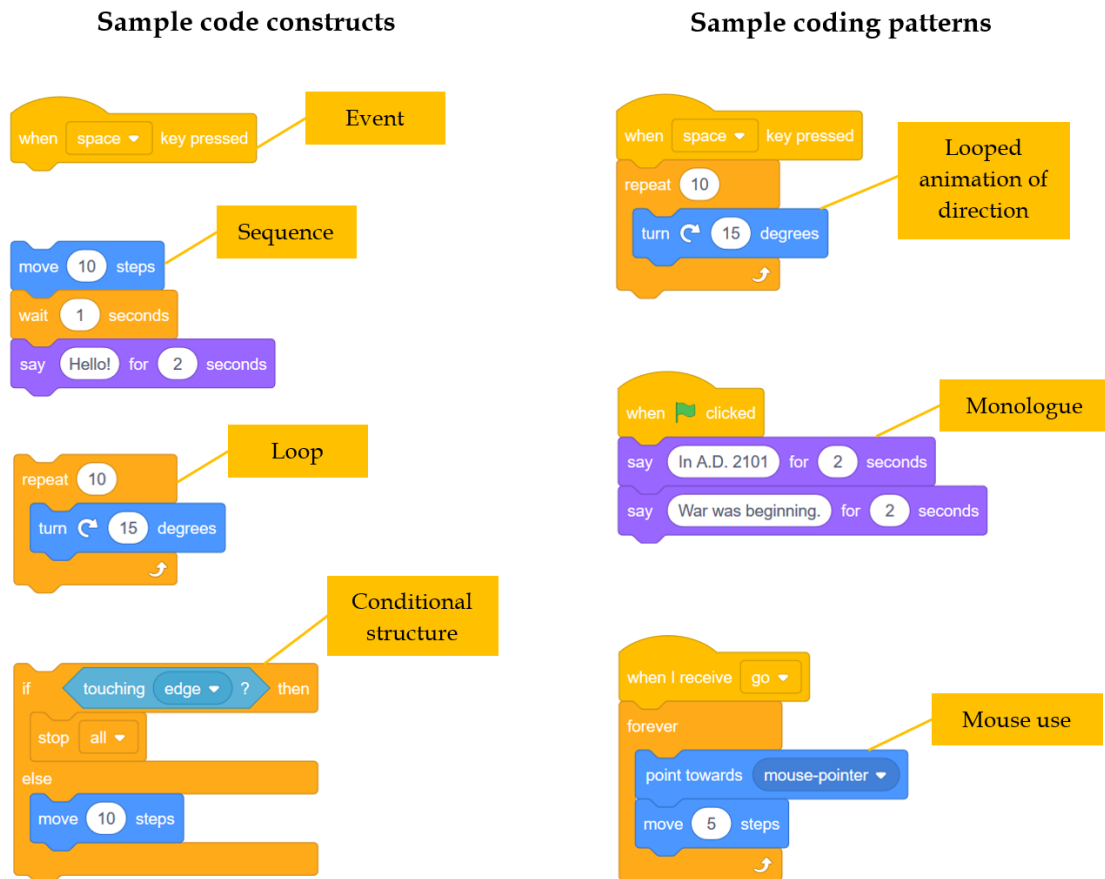


Figure 13. Sample CT-fostering Scratch programming contents

conceptual change to which instructors could match their feedback strategies. This study supported the ideas that students learn specific programming contents more easily than others (see e.g. Seiter & Foreman, 2013) and that specific programming contents are more typical of certain kinds of Scratch projects (see e.g. Moreno-León et al., 2017). The growing knowledge of novice students learning to implement programming contents in Scratch projects may thus justify the proposition of a preliminary, general and overall learning path for CT in Scratch at the primary school level (Figure 14).

To restate the rationale behind the path, specific conceptual encounters with CT appear to occur more habitually among novice learners initially capable of designing more animation-like and storytelling-like projects, which typically encompass such contents as 'timed animations', 'monologues' and 'sprite clicking' (see also Moreno-León et al., 2017; Seiter & Foreman, 2013). Then again, to expedite conceptual encounters with the perhaps more unconventional or advanced contents, it is important to facilitate venturing towards designing more complex games, simulation-like projects and even more sophisticated animations that typically encompass such contents as 'data manipulation' (e.g. in score counting in games), the use of different input and output devices (e.g. with the mouse), 'collision' of sprites and synchronised 'dialogues' among different sprites. The programming course organised in this study (12 sessions) was by all measures short

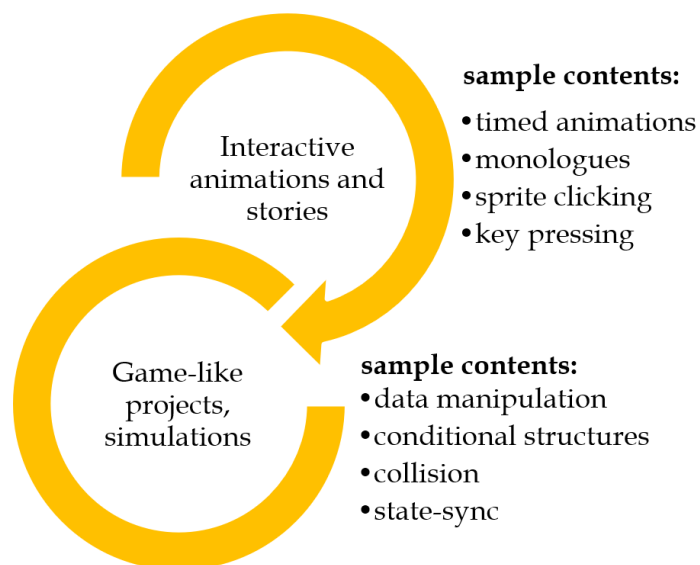


Figure 14. A general learning path for CT-fostering contents in creative coding with Scratch

in terms of venturing towards the more advanced stages along this path, thus advocating for the inclusion of programming over longer periods of time in schools for more profound CT learning.

The sample contents situated in the path are currently merely directive, whereas they could be more methodically categorised for a rigorously developed and validated learning trajectory. In fact, Bloom's taxonomy has been employed previously for defining learning continuums in programming (see e.g. Selby, 2015), but, as Meerbaum-Salant et al. (2013) noted as a limitation in hierarchical taxonomies, some programming contents can be easier to create than others can be to understand. Instead, they developed the Bloom/SOLO taxonomy for particular code constructs (e.g. initialisation, looping). A few other studies (e.g. Gane et al., 2021; Niemelä, 2018; Rich et al., 2018; 2020) have developed focal learning trajectories for particular areas in cross-contextual CT as well. Nonetheless, a meticulous learning progression for all relevant CT-fostering programming contents (in media design with Scratch or other programming environments) at the primary school level would require further development. It may also be relevant to ponder whether very meticulous learning trajectories are needed for the general purpose of introducing primary school students to introductory CT in, above all, an engaging and inspiring way.

Although not formulated as learning trajectories, this study categorised a polymorphous collection of programming contents in Scratch that included the ostensibly variably challenging ways of instantiating the coding patterns as creative features in projects. The abundant contents can hypothetically serve the purpose of providing different learners a suitable level of demand, as highlighted

in formative assessment (Black & Wiliam, 1998; 2009). In the spirit of guided discovery, instructors can facilitate the learning of specific contents uniformly for all students (Mayer, 2004), but this would likely bring the necessity of CTPACK (see Mäkitalo et al., 2019) or another corpus of requisite teacher knowledge into the mix for effective instructional design. Optionally, students can set learning goals for themselves indirectly via creative features (e.g. the coding patterns) in Scratch. This may, in turn, necessitate an increased level of metacognition; however, another solution could be employing a rough learning progression along which students could proceed at their own speed and monitor their progress. Such an approach would align with the formative notion of reinforcing students' performance and self-efficacy by providing them process goals regarding their progress towards set learning goals (Black & Wiliam, 1998).

Following the idea of the 'use-modify-create' model (Franklin et al., 2020a; Lee et al., 2011), learning CT-fostering programming contents could be uniformly modelled as a three-stage progression: 'I can use' (understanding), 'I can modify' (applying), 'I can create' (creating). Such a model has recently been found to provide a productive balance between more structured and more open-ended exploration, to reinforce knowledge by using familiar blocks and to encourage to explore new blocks, combine blocks in new ways and express creatively through creative customisation (Franklin et al., 2020b). The results of Article II also reinforced the ideas that using and modifying pre-existing projects (e.g. debugging challenges, remix projects) and even structured tutorials can lead to learning new programming contents. In turn, pure discovery did not necessarily work well at all, perhaps because it led the students to attempt to create with too many choices without sufficient previous knowledge. On a similar note, Carlborg et al. (2019) concretised how the autonomy in learning (e.g. the amount of available programmatic choices) could be reduced or increased to adjust the difficulty level of learning. Extreme constructionist ideas of pure discovery can be powerful in select situations, but other methods, such as structured tutorials and demonstration, may be occasionally required to accelerate learning in time-constrained classrooms (see also Mayer, 2004; Kirschner et al., 2006).

This study proposes a model for effectuating progressive learning strategies that accounts for different difficulty levels in learning (e.g. for different kinds of learners) (Figure 15). The model is compiled from the results of this study alongside existing pedagogical models on learning introductory programming – the scope of autonomy (Carlborg et al., 2019), the 'use-modify-create' model (Lee et al., 2011) and the 'TIPP&SEE' model (Franklin et al., 2020a). The model is intended for the viewpoint of programming contents as latent learning goals in cross-contextual CT, and it is structured based on current empirical knowledge of students' CT learning in Scratch. The core idea of the model is that a learning strategy (and a consequent level of autonomy and the openness of more exact criteria) is selected for a particular learning scenario focusing on particular programming contents or content areas based on the desired (individual or shared, guided or self-set) level of learning. Importantly, learning may not need to follow the model strictly linearly, and it may not need to always traverse the

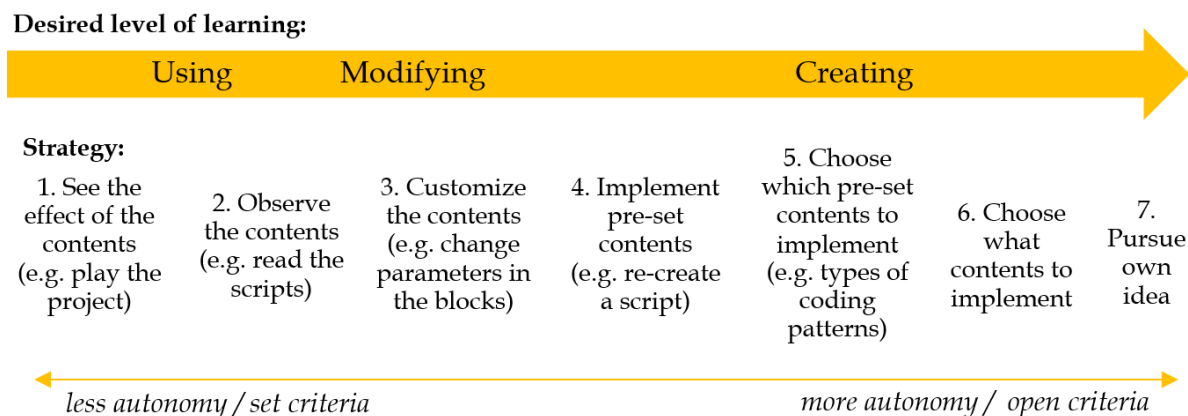


Figure 15. A proposed model for learning CT through manipulating programming contents in Scratch

entire continuum for each content area. Steps can be skipped, as the utility of each step may ultimately depend on a variety of factors, including the previous knowledge and learning strategy adopted by individual students in different situations.

Another key layer in clarifying learning intentions for CT through programming is that of CT-fostering programming activities. The programming activities could be considered as design strategies or procedural skills aside contents that are implemented in programmed projects (Angeli et al., 2016; Csizmadia et al., 2015; Grover & Pea, 2018; Shute et al., 2017). In general, they may be altogether more relevant in more design-like situations (i.e. at the 'create' stages in the learning processes) as indicated by their core definitions: planning in CT should effectively guide the design of computational solutions (Csizmadia et al., 2015), and iterative design should be valuable for the expectation of refining the artefact through cyclical development (Brennan & Resnick, 2012). Then again, collaboration can be emphasised for its capacity to enhance knowledge construction (Roschelle & Teasley, 1995), while testing and debugging can ensure that the designed solutions achieve what they intend to achieve (Ben-Ari, 1998).

This study showed how the CT-fostering programming activities can manifest as behavioural actions in pair programming in Scratch in addition to what actions were more common, how they positioned temporally and how specific challenges appeared amidst them. Based on the nascent empirical results obtained in this study and the scarcity of previous empirical research, it is not entirely straightforward to define what kinds of specific actions are conclusively effective or ineffective either universally or at specific moments of learning. Therefore, it still remains slightly unclear how the effectuation of particular 'good' actions or the avoidance of 'bad' actions should be justifiably posed as common goals in CT. Instead, current best knowledge is that different kinds of actions are situationally relevant, and it can be favorable for students to become aware of ways of putting the activities into practice that can be useful in particular situations. Current knowledge can thus be incipiently synthesised to nominate actions that can be profitable for programming and learning CT (Table 3).

Table 3. An incipient synthesis of profitable actions in (i.e. potentially good ways for) carrying out CT-fostering programming activities in Scratch

<b>CT-fostering programming activity</b>	<b>Profitable actions</b>
Project planning	Drawing pictures of sprites and backdrops Ensuring a programmatic core by planning with familiar contents (e.g. with block-like pseudocode) Opening new learning opportunities by planning with imagination (e.g. with human language)
Iterative design	Allocating time-wise suitable phases for the design process (e.g. graphical design and coding) Remaining focused on the task Making reasoned (i.e. not arbitrary) design attempts or modifications Becoming familiar with the practical affordances in the programming environment (i.e. the different code blocks and editing tools) Paying attention to computational similarities in the scripts to generalise abstractions of contents
Collaboration	Activating the driver and the navigator in all design events and phases Discussing ideas beforehand and making bilateral decisions Seeking help from peers and instructors Browsing and remixing existing projects and searching for information on the Internet
Testing and debugging	Testing play often after coding Preparing to deal with uncertainty and seek information Staying conscious of the purpose of and contents in each script Avoiding blind guessing

### 6.2.2.2 Evincing student understanding

The assessment rubrics used in Articles II and III can denote students' conceptual understanding in CT, such as the knowledge required for instantiating different types of coding patterns, and their practical skills in CT, such as their ability to fix bugs effectively. An essential purpose of evincing student understanding is to harness information gained from the learning activities to move learning forwards in line with the learning goals (Black & Wiliam, 1998; 2009). For this purpose, learning the various conceptual and practical areas in CT could be examined through the phenomenon of conceptual change (see Duit & Treagust, 2003).

However, when viewing CT as a competence learnt through programming (Tang et al., 2020), conceptual change in CT is bound to be examined as a latent phenomenon. To do so, similar to how the rubrics for programming contents and activities in Scratch can be used for setting indirect learning goals for CT, student



understanding in CT can be evinced through contents in their programming projects and the programming activities they carry out while programming (Grover & Pea, 2013; Román-González et al., 2019; Seiter & Foreman, 2013). As a concrete example, dysfunctionally implemented programming contents (typically bugs) can be valuable demonstrations of naïve knowledge (e.g. misconceptions) – conceptual understanding that is expected to change. Similarly, inefficient or erratic actions (e.g. arbitrary modifications or guessing while fixing bugs) may similarly imply the lack of established skills (see also Ben-Ari, 1998; Swidan et al., 2018). Conceptual change in CT through programming could thus be viewed through practical exhibitions of the realisations as to how particular programming contents achieve specific intended creative features and how specific actions are effective for improving the design.

On the whole, more traditional assessments, such as aptitude tests (e.g. Román-González et al., 2017a), may be established in more stringent validation processes and thus be more reliable in the pursuit of evincing student understanding in CT. Their use may nonetheless guide more towards diagnostic or summative purposes than formative ones (Keeney, 2008). The assessment rubrics employed in this study and intended for formative use were founded upon a diligent literature review (in Article I) and studious methodological development (in Articles II and III). Their development intended to translate learning goals into understandable terms (Black & Wiliam, 2009), improve validity in observation-based assessment (Seiter & Foreman, 2013) and enable the analysis of detailed programmatic targets to facilitate accurate feedback (Hao et al., 2021; Vihavainen et al., 2013). Altogether, this study showed support for the claim that different analyses can provide important information for an overall view of students' CT (Basso et al., 2018; Grover et al., 2017).

Evincing conceptual change through contents in projects or activities during programming processes can be performed in different ways. As shown by earlier research and also as carried out in this study, the analyses can be based on, for instance, presence (e.g. Burke, 2012), frequency (e.g. Maloney et al., 2008), correct implementation (e.g. Seiter, 2013), completion rate (Franklin et al., 2013) or comparison to success criteria (e.g. Sáez-López et al., 2016) in more or less quantitative or qualitative ways. With this study's understanding, universal premises in assessing student understanding could be phrased as the observment of 'functional and purposefully implemented programming contents' and 'efficient and purposefully effectuated programming activities or actions'. What is purposeful can depend on the context of learning – what the project being designed intends to achieve. Different analysis methods may therefore be appropriate with respect to the project type, learning goal and level of learning (see Figures 13 and 14). More tangibly, the evincing process can be shaped by such questions as 'To what extent does the project contain these desired contents?', 'What/which of these contents does the project contain?' and 'How does this action improve the design of this project?'

Evincing reliable evidence of conceptual change can require careful longitudinal assessment. The results show that although conceptual change in CT can

be fostered by implementing contents and effectuating activities, single (or even multiple) conceptual or practical encounters may not ensure that the contents or activities in question are subsequently always put into practice functionally or steadily (i.e. learnt well). The implementation of specific contents or the effectuation of specific actions may also have been a result of a coincidence or caused by external factors (e.g. teacher solving the problem), demonstrating how assessments of mere end products can be questioned (Lye & Koh, 2014). Altogether, it is justifiable to postulate that continuous conceptual and practical encounters are more reliable in implying conceptual change, especially if the implementation of contents or the effectuation of actions occurs autonomously in new situations. Similar arguments also appear in learning progression taxonomies in which the ability to abstract computational models and create new programs in programmatic situations that vary in complexity can denote higher-order learning (see e.g. Meerbaum-Salant et al., 2013; Selby, 2015). In all, repetitions in implementing contents and effectuating programming activities functionally, effectively and purposefully (i.e. in a self-regulated way) can be crucial for both learning and eliciting valid evidence of learning.

Moreover, importantly, evincing student understanding in 'all of CT' through all possible kinds of programming contents and activities can be an overwhelming task. Guidelines to direct the focus of manual assessment to essential targets may be in order. For this purpose, the results of this study suggest there are specific areas in CT that students grasp more autonomously – thus also appearing to need less support for learning – while other areas, such as specific programming contents, design-related actions and social actions, may benefit from more deliberate guidance.

The results justify the synthesis of essential **target areas to evince student understanding** in introductory CT through Scratch among novice learners at the primary school level (Table 4). The targets (both content and activity-oriented), which are by no means exhaustive based on this exploratory study, are formulated especially from fundamental or profitable CT-related areas that the students did not necessarily learn autonomously or effectively in Scratch. However, although the students appeared to learn to implement specific programming contents (e.g. the 'Animation' coding pattern, 'looping') or practices (e.g. testing play often) more autonomously, all skills and knowledge likely need to be learnt as purposefully as any other ones.

Another element that can burden the practice of evincing student understanding in CT in programming is the process of the evincing itself, particularly when considering the traditional approach of the teacher as the manual assessor. Complicated research-designated tools as employed in this study are labour-intensive and time-consuming. As emphasised in curricular guidelines (e.g. Opetushallitus, 2014) and contemporary pedagogical literature (Black & Wiliam, 2009), programming also presents – and perhaps even encourages – the practices of peer assessment and self-assessment to promote social constructivist learning and metacognition. Additionally, there have been efforts to automatise assessments in Scratch. Currently, a tool called Dr. Scratch provides summative scores

for Scratch projects based on the presence of particular code constructs (Moreno-León et al., 2015). A discontinued automated assessment tool called Scrape provided a visualisation of code blocks used by a programmer in Scratch (Brennan & Resnick, 2012). Such tools could be modified to provide students a self-assessment dashboard to monitor some conceptual encounters with CT over time. Similarly, a tool for reviewing the process of implementing code blocks in scripts (see Funke & Geldreich, 2017) can provide a fruitful opportunity for self-reflection of design processes. Moreover, the broadly used *Creative Computing* guide for teachers (Brennan et al., 2014) presents such methods as ‘critique groups’, ‘project pitching’, ‘unfocus groups’ and ‘gallery walks’ that can be structured to focus on the assessment of specific CT-fostering contents or activities.

Table 4. Essential target areas to evince students’ introductory skills and understanding in CT through Scratch at the primary school level

Target of assessment for learning	Specification
Fundamental programming contents	The basics of scripting, namely implementation of the ‘control’ and ‘coordination’ code constructs
Advanced programming contents	Implementation of more advanced contents, such as ‘data manipulation’, ‘collision’, ‘conditional logic’ and ‘custom variables’
Recurring implementation of functional contents	Steadiness of efficient design as observable through, for instance, the lack of recurring bugs
Creative planning for constraints	Accounting for, for instance, previous skills, the intended level of learning and the allocated time
Knowledge of code blocks and editing tools	Awareness of what tools to use and selections to make
Generalising solutions	Understanding of algorithmic similarities in types of coding patterns and code constructs (e.g. ‘initialisation’)
Comprehension of the scripts	Cognizance of the functional parts of one’s own project for efficient project refinement and location of errors
Making aforethought design modifications	Avoiding blind guessing
Bilateral decision making	Constructive talk and equal participation
Information searching	Activating peers in the classroom and searching for information or materials on the Internet

### 6.2.2.3 Providing feedback

This study provided insight of ways to provide feedback for CT learning in Scratch in the classroom. The purpose of feedback is to move learning from its current (evinced) state towards the learning goals (Black & Wiliam, 2009). It is important to begin with noting that constructionism emphasises adapting to versatile learning strategies by personalising learning and promoting active

searching and discovery. The paradigm underscores the increased effect for the relevance of learning when new knowledge (e.g. a new conceptual area) emerges 'organically' when it is truly needed (Brennan & Resnick, 2012; Resnick et al., 2009). Such moments can be extremely powerful for learning when they manifest through the self-directed, active externalisation of one's own thinking (e.g. designing a program) with a computer (Papert, 1980). In a sense, the computer (or the programming environment) is thus always present to provide 'feedback', not least in the form of bugs (Ben-Ari, 1998); the programmer has implemented an erroneous computational model, and the computer states this by behaving in an unexpected manner. Therefore, there can be room for students to build their own learning processes in programming by solving problems more or less alone.

Decades of research on learning by constructivist ideas does not, however, support the notion of leaving the students completely alone. In fact, prior research supports the better impact of strong instructional guidance, especially among novice learners (Kirschner et al., 2006). A central problem in pure discovery is students' inability to select relevant incoming information. Support can be necessary in programming to help students make sense of the information at hand, organise it and integrate it with previous knowledge (Mayer, 2004). In practice, problems in programming can manifest as the inability to understand what actually went wrong when a computer states there is a problem. In other words, students may not make sense of what mental model conflicts with the brutal feedback from the computer (Ben-Ari, 1998). Such findings were commonplace also among the students programming with Scratch in this study. Therefore, this study understands that, as underlined in formative assessment (Black & William, 1998; 2009), intervention in the form of providing deliberate feedback can be highly beneficial among school students while pair programming in Scratch, akin to how deliberate instructional design is carried out in the clarification of learning goals and evincing of student understanding to gain goal-oriented information to enhance learning.

Feedback in programming can be versatile. In fact, social constructivism states that knowledge can be built through social interactions (Roschelle & Teasley, 1995). Key ways to retain the foremost strengths of constructionism (i.e. active searching and discovery) and incorporate the utility of feedback can be to promote shared knowledge construction through pair programming, peer interactions and searching for information on the Internet (Arisholm et al., 2007; Brennan, 2013; Brennan & Resnick, 2012). In theory, students working together towards a shared goal may naturally give constant feedback to each other while building mutual understanding. In turn, information sharing and receiving feedback can occur informally in the classroom or through facilitated moments, such as 'gallery walks' (Brennan et al., 2014). Then again, the online Scratch community holds millions of projects available for reflecting on one's own work (Brennan & Resnick, 2012).

Despite the voiced benefits of social interactions in programming, this study was unable to excavate very rich empirical evidence of them. The study merely showed further evidence that novice programmer students can carry out such

acts as negotiating and suggestion making in programming (see also Campe et al., 2020); however, the role of such moments as potential mutual feedback in the process of shared knowledge construction remained unexplored. Perhaps more importantly, the students demonstrated practices that may even have been adverse to discovery and collaboration. In short, the students were not always able to make sense of new conceptual areas (e.g. code constructs), find new programmatic possibilities and put them into use and activate each others' knowledge in the shared work equally and effectively. Even more, the results did not reveal ways in which knowledge construction beyond the computer could have operated as mirrors for one's own work because such events were scarce or even non-existent. In conclusion, the results suggest that school students' collaborative CT learning processes in Scratch could benefit from or even necessitate scaffolding, that is, planned or spontaneous support tailored for individual learners to accelerate learning (see also Touretzky et al., 2013; Vihavainen et al., 2013).

Solutions in **planned scaffolding**, which may not manifest as feedback *per se*, can occur in the ways elaborated in the previous sections. To recap, they can include clarifying focused and progressively challenging learning goals in sensible and interesting ways to maintain students' motivation and keep them focused on the task (see also Lye & Koh, 2014). As a form of 'self-feedback' (reflection), students can monitor their learning progress through, for example, a project portfolio or review their previous programming processes to contemplate what could be done differently. Additionally, important acts of planned scaffolding can include those that the results of this study implied to perhaps have acted against learning, including building the physical environment to support collaborative practices (e.g. equal access to the input control devices), pairing creatively or attitudinally like-minded students or those having similar learning strategies and ensuring a common knowledge background (Ally et al., 2005; Scherer et al., 2018). Problematically, different solutions may work very differently for each individual learner (Denner et al., 2014).

Solutions in **spontaneous scaffolding** during programming processes could, in turn, be interpreted as ways of providing timely and targeted feedback to move learning forwards. Research regarding effective acts of guidance in Scratch is only taking its first steps, and this study provided little insight on top of the growing body of knowledge.

First, although the effectiveness of different guidance acts cannot be generalised yet, levels of help-seeking, varying from asking for validation to 'how' to implement something (see also Franklin et al., 2013), and help-receiving, varying from instrumental (e.g. demonstrating which similar coding pattern already works) or executive (e.g. stating what code blocks are required) help (see also Mäkitalo et al., 2011), were beneficial depending on the students' particular situation. This is perhaps expected given the complexity of the conceptual and practical areas manifesting in CT next to the students' presumably varying skill and knowledge levels, thus inevitably complicating the development of an all-fitting pedagogical model of effective feedback. Perhaps, in principle, more instrumental or implicative help guides students to process information more themselves,

as emphasised in socio-constructivist learning (Roschelle & Teasley, 1995). However, as demonstrated by the data, sometimes executive or explicative help can be necessary (Kirschner et al., 2006), for instance, to reveal entirely new conceptual areas, code blocks, design tools or effective design acts or social acts.

Second, the results of this study illuminated concrete forms of help-receiving that, alongside previous research in guidance, can be used to advance knowledge of appropriate ways of providing feedback for learning CT through implementing programming contents and effectuating programming activities in Scratch (Figure 16). These ways of feedback can be plural. They include demonstrating a solution to the problem, for instance, in terms of how to break the intended creative functionality into coding patterns and code constructs. Students' attention can also be directed to focal places, such as pre-existing functional solutions (e.g. initialisations) or relevant information (e.g. parameters) in the implemented contents (Lye & Koh, 2014). Asking elaborative questions may prompt reflection and guide attention to relevant contents, such as locations of bugs (Webb & Rosson, 2013). Available choices regarding, for example, which coding patterns or code constructs could suit the planned feature, can be narrowed (Carlborg et al., 2019). Altogether, information can be presented in different ways, including with pseudo-code and metaphors, such as a hand-mixer for 'looping' (Pérez-Marín et al., 2020), road junctions for conditionals (Georgina et al., 2015) or embodied metaphors with gestures (Manches et al., 2020). Milestones for, for example, how much of the pre-set contents have been implemented may keep students on the right path (Luo, 2005; Nickerson et al., 2015). In addition, other known ways, such as reminding of the time left, repeating previous learning and conceptualising students' thinking (using the vocabulary), may be helpful.

Decisive requisites in providing efficient feedback can be sufficient for communicating relevant information in addition to appropriate knowledge for selecting favorable feedback methods at the right time. Routine pedagogical knowledge may partially suffice; however, for learners, a key purpose of obtaining feedback is getting in touch with expert knowledge (Kirschner et al., 2006). A teacher can be an expert in CT (Mäkitalo et al., 2019), but given the current state

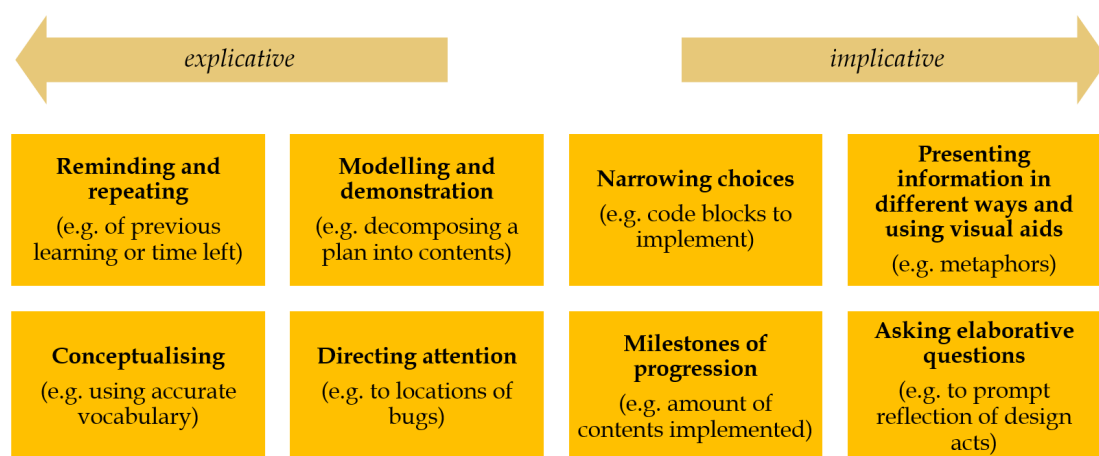


Figure 16. Ways of providing spontaneous feedback for students programming in Scratch

of the inclusion of CT and programming in schools (see e.g. Fraillon et al., 2020; Kaarakainen et al., 2017), this may rarely be the case. Yet, expert knowledge can reside elsewhere too, such as in the Scratch community (Brennan & Resnick, 2012) or elsewhere in the educational community, such as among more experienced student tutors in the school (Korhonen & Lavonen, 2016). Teachers may not need to know everything if time is available to consider the options and seek information from various sources.

The lack of time typical of schools may be an obstacle for facilitating the giving and receiving of efficient feedback. Further support for spontaneous acts of feedback on students' programming work can be necessary. This study developed concrete rubrics for CT-fostering programming contents and activities. All the discussed formative assessment processes can point towards dealing with this substance; for instance, the segments that can be clarified as learning goals and evinced as latent indications of student understanding, manifesting as programmed contents in projects or effectuated programming activities, can operate as targets of in-time targeted feedback. As intended in the development of the rubrics for programming contents, there is an opportunity to reveal bugs through dysfunctional contents as examined in Article II, potentially denoting manifestations of misconceptions, and to facilitate further learning, such as by illustrating new types of coding patterns that are similar to the ones currently implemented (Figure 17). Similarly, the analysis of students' programming activities can reveal what undesired actions are occurring, and feedback can concern what actions could replace them.

Convenient and systematic instrumentalisation of feedback for programming contents and activities would require practical tools or models. Automated assessment tools have been proposed and used in other programming environments as ways of providing well-timed contextual feedback, particularly for programming projects (Hao et al., 2021; Vihavainen et al., 2013). Such tools have enabled code analysis, especially as a mode of self-assessment in Scratch and for particular areas in CT (see e.g. Moreno-León et al., 2015). Recently, Talbot et al. (2020) reported on the automatic assessment of patterns in students' Scratch programs, which provides reason to assume that areas in the rubrics used manually in this study could be automatised as well.

Importantly, however, there are aspects in CT that computerised feedback providers may not recognise. In particular, it may be difficult for them to determine what conflicting mental model the students have that is causing a particular bug and what instructional strategy fits the learners and the learning situation in question, thus being perhaps constrained to provide more or less generic feedback. Additionally, as noted in Article I, not all areas in CT are indicated by static contents in projects, such as those with relevance to events beyond the computer (e.g. social actions). The indications for CT can also be qualitative in nature (e.g. precise modification sprites' parameters, such as size) or computationally complex, especially in advanced projects, leaving much of the important feedback on the shoulders of human-to-human interaction.

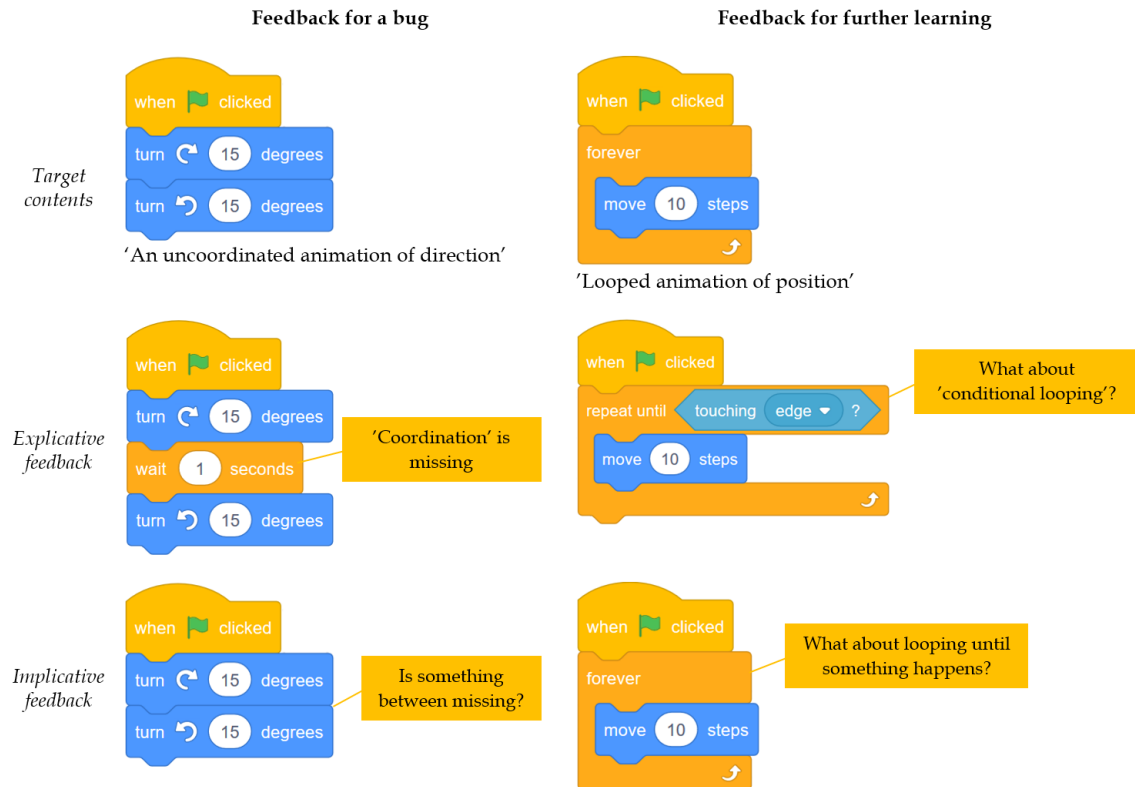


Figure 17. Hypothetical feedback for CT-fostering programming contents in Scratch projects

Formative assessment should altogether be considered a situational and a contextual process rather than a purely technical one. It essentially includes, for instance, observing students' demeanour, initiative taking and explication of key substance. The quality of teacher-student dialogue during formative assessment and the reception of and response to feedback by students may involve intricate personal and social factors, such as self-perception, teacher's beliefs, student's attitudes to learning and such emotions as insecurity and fear of failing. Moreover, as students must be aware how far away they are from their objectives and how they can reach them, metacognitive skills in, for example, perceiving the level of one's own understanding and the gaps in it play a key role (Black & William, 1998; 2009; Keeley, 2008). Automated assessment should therefore perhaps be considered not as a replacement but as a supplement to teachers' pedagogical expertise and the traditional, valuable student-teacher discourse.

Last, the various challenges that students can face while learning CT in programming (e.g. lack of success in debugging, social conflicts, not finishing in time) can be complex, very cumbersome and even discouraging. Still, they should not be considered purely negative. For instance, bugs can pinpoint where naïve conceptual understanding exists (Ben-Ari, 1998) and provide an opportunity to learn meta-skills (e.g. dealing with uncertainty, searching for information to solve the problem) (Barr & Stephenson, 2011). The purpose of formal education is to facilitate learning and growth, and, especially in constructionist-focused hands-on activities, encountered challenges can show where they could importantly take



place. Therefore, not facing any bugs can perhaps be considered a rather wasteful learning process in programming.

### 6.3 Limitations of the study

This study acknowledges particular limitations that concern the current overall knowledge skirting the investigated topic in addition to the research design, that is, the employed methods and the collected data. The limitations and their potential influence regarding the obtained results are disclosed in this section, and the gaps that the limitations have left are discussed as avenues for further research in the subsequent section.

#### 6.3.1 Investigating CT

Specific aspects regarding the author's professional understanding of the topic should be discussed to ensure the transparency of the reported observations and reasoning in this study. The author is a Master of Education by formal training, and his professional understanding of computing and CS is limited to hobby-level programming and bachelor-level studies in educational technology (including CS1 and CS2 courses). The author's understanding of CT as an educational topic has mainly influenced the thesis process (i.e. completion of the three articles). In particular, after beginning this research project, the main theoretical concepts in CT were determined through a thorough literature review to reinforce external and code construct validity (Yin, 2012). Subsequently, the author participated in national conferences on education and educational technology in addition to international conferences and doctoral consortia on CT and computing education. The view of CT in educational practice has been pivotally influenced by the implemented programming course and the author's experiences in teacher training at the university level and in several national in-service training projects and school projects in the Innokas Network.

The interpretations or choices made in this thesis can be dissected in multiple ways and to varying levels of granularity. One way to discuss the interpretations can be through the key places where there is lack of consensus in CT. In all, the shortcomings in clearly defining CT justifies a fair amount rightful criticism in the topic. Despite such threats that emerge from these notions to the salience of CT, they can be viewed as facilitators of the maturation of this topic.

The first major domain of criticism has to do with the motifs of introducing CT in schools. One such motif is preparing students to work in the ICT sector. This study restates Mertala et al. (2020) in the argument that although such an economic or even political uplift includes the need to have the female population better represented in coding professions, there are optimistic and, in certain senses, misleading claims around maintaining political economy and competitiveness. In fact, studies conducted in several countries have recognised that pri-

vate and multinational corporations have influenced the instatement of programming-related educational reforms and have even carried out comparably large-scale teacher training interventions. The manner in which financial and capitalistic demands have shaped educational policy and practice has perhaps been rightfully criticised (Mertala et al., 2020). In all, this study viewed that a justified motif of introducing CT in schools can be a combination of several arguments and therefore focused on investigating CT from a clearly stated viewpoint (elucidated in section 2.1).

In addition, it is not entirely straightforward to define the skill and knowledge areas that all students should acquire to become sufficiently skilled computational thinkers and computational problem solvers for subsequent studies and work life. In fact, scientific efforts from the last decade have only brainstormed what CT can teach exactly. The theoretical notions and the literature review carried out in this study are founded largely upon rationalisation of the existing knowledge that, according to a recent scientometric study, has a strong cultural inclination to the US (Saqr et al., 2021). Critically put, there is reason to suspect that the current state the topic is under the influence of only developing theories and culturally biased educational ideas. This study attempted to mitigate any potential effect caused by the maturing state of CT by critically contemplating the adequacy of the current theories in the investigated educational context (i.e. Finnish education).

A related matter has to do with the incompleteness in defining CT compared to other types of thinking<sup>22</sup> and its position alongside other curricular topics, such as technology education, media education and civics, which have been used to target the acquisition of similar skills (Kalelioğlu et al., 2016). In fact, CT in terms of its ‘computational problem solving’ aspect could be interpreted as a combination of certain ways of thinking and ICT skills, which are portrayed in various renderings of the 21<sup>st</sup> century skills (e.g. Binkley et al., 2012) and the current Finnish primary school core curriculum (Opetushallitus, 2014). Similarly, the ‘computational literacy’ aspect of CT could be viewed as a specific thematic branch in the ‘multiliteracy’ topic, which is also depicted in these frameworks. Meanwhile, some scholars (e.g. Sengupta et al., 2018) posit that programming in K–12 should be reframed more generally as ‘modelling’, which could help educators better adopt it across subjects and disciplines. It could also be interesting to consider whether a focus on programming would be enough for the kinds of

---

<sup>22</sup> CT and programming skills have correlated positively with, for example:

- reasoning and mathematical skills, such as modelling and data analysis (Popat & Starkey, 2018; Scherer et al., 2018; Shute et al., 2017),
- critical thinking (Popat & Starkey, 2018),
- creativity (Durak & Saritepeci, 2018; Israel-Fishelson et al., 2020; Scherer et al., 2018),
- social skills and self-management (Popat & Starkey, 2018),
- computer and information literacy (Fraillon et al., 2020),
- algorithmic thinking, cooperation and critical thinking (Durak & Saritepeci, 2018),
- general problem solving skills (Çiftci & Bildiren, 2020; Durak & Saritepeci, 2018),
- mental rotation skills (Città et al., 2019),
- nonverbal-visuospatial reasoning, arithmetic abilities and different aspects of numeracy (e.g. fact retrieval and problem completion) (Tsarava et al., 2019)

objectives that CT education as portrayed in this study is expected to reach. Is CT just a trendy term promoting programming education, which, in fact, covers all fundamental computational skills that 21<sup>st</sup> century learners need? These kinds of conceptual forks currently spread the tentacles of CT far and wide to other (perhaps more well-established) disciplinary frameworks and types of thinking, further confusing educational experts' views of CT. Amidst this labyrinth, this study attempted to synthesise an inclusive but theoretically justified collection of what core skills and knowledge CT encompasses.

A third criticism in CT relates to its somewhat optimistically presumed built-in transferability. CT is often comprehended as a fundamental skill for everyone regardless of their work domain or study discipline (Barr & Stephenson, 2011; Lu & Fletcher, 2009; Wing, 2006). The integration of CT within and across disciplines has been examined in many prior works (e.g. Basu et al., 2014; Israel et al., 2015; Niemelä, 2018; Perković et al., 2010; Sengupta et al., 2013; Settle & Perković, 2010; Weintrop et al., 2015; Yadav et al., 2016). In short, the notion of universally applicable CT that is necessary for everyone has received some criticism, most notably due to the lack of empirical support (e.g. Denning, 2017). Guzdial (2015, p. 60) pointed out that 'there has not been a study since Wing's 2006 paper that has successfully demonstrated that students in a computer science class transferred knowledge from that class into their daily lives.'

However, it is essential to disclose what is the kind of transfer that CT should achieve. In one sense, learning CT or programming has not been shown to teach generic problem-solving skills (De Bruyckere et al., 2020, p. 7–10). However, the significance of CT does not depend on whether it fosters the learning of skills far beyond its own disciplinary essence or not; instead, it can be justified (and has been justified herein) to be important in its own right. However, the more important kind of transfer is perhaps whether learning and utilising CT in different computational contexts, such as in different programming environments in schools, transfers to other computational contexts, thus providing rationale for the existence of cross-contextual CT skills altogether. Do algorithms for storytelling in Scratch teach you to design algorithms for professional simulations?

To be precise, the transfer of skills from programming domains to non-programming domains and the acquisition of general cognitive skills hypothetically gained through programming has been critically discussed for decades (Klahr & Carver, 1988; Pea & Kurland, 1984; Salomon & Perkins, 1985). Empirical studies have demonstrated both positive transfer effects between programming languages (e.g. Armoni et al., 2015; Wu & Anderson, 1990) and minimal to no transfer effects (e.g. Parsons & Haden, 2007). They have also highlighted cognitive parts that can facilitate the transfer of certain types of computational content (e.g. Franklin et al., 2016). A relatively recent meta-analysis on transfer effects in learning programming (Scherer et al., 2018) showed evidence of a strong effect for near transfer and a moderate effect for far transfer.

Studies of transfer in CT, a higher-order competence that programming should foster, are lacking (Hsu et al., 2018). The current best knowledge on transfer in CT suggests that the contexts of learning and subsequent application should be similar (Bull et al., 2020). Denning (2017) saw that the controversial claims of CT's sweeping transferability can produce obscure definitions for it, and teaching it in classrooms may therefore turn out to be little more than a struggle for teachers. The same problem perhaps goes for other relatively ambiguously described 21st century skills, which are still quite hard to pinpoint and conclusively define to investigate their meanings very rigorously. Therefore, this study is also left to rely on the prospect rather than the assurance that the investigated kind of CT transfers to other computational contexts. Restraint was thus exercised in assuming any kind of assured transferability of the CT studied herein; rather, this matter is clearly stated as one of the most crucial topics of further research.

### 6.3.2 Case study with Scratch

The empirical side of this study is subject to the preconditions, advantages and deficits of a case study design (Yin, 2012). In short, the case study could not (and did not aim to) gather data to make systematic comparisons, find trends and generalise the findings to different teaching and learning situations. Instead, it aimed to dive into the particular case deeply, apply novel theoretical and methodological approaches and uncover in-depth knowledge by composing rich accounts of the students and their encounters with CT and programming. The results were primarily targeted for wider comparison, creation of theoretical models, stimulation of hypotheses for experimentation and further methodological development. The selection of the participants employed convenience sampling, that is, drawing the sample non-randomly with emphasis on practicability instead of randomness. The results may therefore not represent the population thoroughly, which is why care was exercised when deriving generalisable implications from the results.

Scratch was selected as the practical context to be examined in the study for the practical reason that it is free, accessible, multilingual, versatile and has a 'low floor' (i.e. that it is novice-friendly). It is still only one programming environment among many, and it has a one-of-a-kind quality and perhaps even pedagogical ideology in creative computing (Brennan & Resnick, 2012). For instance, educational robotics and microchip tinkering (e.g. Micro:bit, Arduino) – which have in part also been profoundly 'hot', especially in the context of STEAM education – promote activities that are different in nature when compared to designing interactive media projects. The author's view is, however, that despite their far-reaching educational benefits in facilitating, for instance, collaboration, hands-on doing and being immersed in physical devices, maker learning contexts may have less expansive and slightly narrow connections with CT (e.g. a focus on mechanical and electrical engineering) than programming contexts with an aim of producing purely digital artefacts. Additionally, despite the relatively high level of

digital equipment in Finnish schools, technology is expensive, and free of charge ways to open doors to design-oriented CT are more equitable.

It is also possible that newer technologies, such as those focusing more on, for example, artificially intelligent systems and less on ‘finite automata’ machines and imperative algorithmic principles, can turn up to offer new kinds of tools, gadgets and computational kits with which CT is practiced in schools. Although the results are applicable mainly in Scratch, the expectation of this study is that the theoretical foundations would be analytically generalisable to other problem solving domains.

### 6.3.3 Observational methods

Limitations in the central methodological approach of this study—assessing students’ CT by means of observation—concern the two main analysis methods used, artefact analysis and programming process analysis. In essence, reliability and validity in assessing CT through programming are metrics for how much we can trust what assessment tells us (Tang et al., 2020). This issue is especially important when assessment is carried out by methods of observation, for which only a few reports of reliability and validity by, for instance, comparing scoring results with other validated tools such as inter-rater reliability, expert judgement and exploratory factor analysis exist (Jun et al., 2014; Tsarava et al., 2018). The lack of the psychometric reliability of validity in assessing CT-related topics is a common issue in learning CT through programming; for instance, Lin et al. (2020) reviewed 60 STEM-related maker activity assessments and found that only 15% provided evidence of reliability and validity. Furthermore, it is also possible that assessment instruments that have been ‘validated’ by comparing them to previous instruments that have been subjected to the same process may merely accelerate the ‘downward spiral’ of invalidity.

An overall limitation in this study is that no tests, questionnaires or other assessment methods were employed to compare findings regarding the students’ learning obtained via the artefact and process analyses. This was due to the limited time to complete the dissertation while focusing on the main goals of this study. The development of the analyses relied strongly on existing research, and other available measures were taken to increase validity and reliability (see details below), but they did not undergo rigorous validation processes. Such processes could have specified their operational underpinnings and contributed to their capacity in reliably indicating students’ actual CT capabilities, potentially leading to stronger and more reliable evidence. However, this limitation was considered acceptable in this early-stage study and case study design, which did not aim to produce generalisable results for large-scale decision-making.

A related limitation in this study concerns the validity in determining participants’ cognition based on the results of their hands-on work. The artefacts that the students programmed were only latent measurements of their thinking (Seiter & Foreman, 2013). In particular, the block-based programming environments in which students can drag-and-drop pre-existing blocks to form scripts incorporate a risk of students designing something that they do not understand

(Lye & Koh, 2014). Although research investigating and remedying the reliability and validity issues in artefact analyses has begun to emerge (e.g. Román-González et al., 2019), this issue still remains worthy of scrutiny.

To advocate the validity of programming contents evinced as indications of thinking, it is preferable to consider the context of the contents (e.g. recognising the learning assignment) and examine meaningful evidence semantically rather than merely technically (Seiter & Foreman, 2013). The artefact analysis in Article II increased construct validity (i.e. investigating the intended competence) specifically in this way. The semantics-directed patterns-first evincing process of programming contents can be justified to produce a more legitimate manifestation of student understanding compared to assessments that are merely concerned with the mere 'nut and bolt' qualities in programmed projects. Furthermore, the artefact analysis employed a rigid rule-based analysis to decrease the requirement of making qualitative interpretations.

Despite measures taken to combat issues of validity and reliability in artefact analysis, it is important to note how products come to be (Lye & Koh, 2014). Although triangulation methods were not employed systematically to account for increased reliability in all findings, this study complemented artefact analysis with a sample of process analyses and discussed the insight gained in the overall examination of the findings. Additionally, grounded by the fact that students' CT activities are not necessarily directly identifiable from their programming processes, the multilayered analysis of programming activities can be justified to encompass comparably high construct validity. The process analysis also incorporated a blind inter-rater reliability check to increase reliability in the analysis.

The analysis of the students' shared programming processes resulted in additional potential limitations as well. In pair programming – although pedagogically meaningful – it can be difficult to estimate the level of understanding of each individual learner. Moreover, real-life incidents in the non-controlled classroom situations may have masked or distorted key evidence regarding the students' capabilities. Perhaps most fundamentally, the audio data turned out to be partially unclear due to the noisy classroom atmosphere, resulting in discourse analyses being limited to a rather generic level. Additionally, the students' programming activities may have been influenced by contextual and non-programmatic factors that were possibly not caught by the recording devices. In summary, although the depth of discovery was not as deep as possible, this cost was acceptable in order to acquire evidence from authentic classroom situations. The evidence obtained to speak to the students' learning in CT (rather than their conceptual and practical encounters as examined herein), however, may not be complete and exact.

## 6.4 Future research

This study raised several fruitful opportunities for future research on this young and unestablished topic. The opportunities stem from the possibility of deepening the insight gained in the study by using the employed methods in novel ways or accounting for the limitations. They also stem from the opportunities to adopt different theoretical viewpoints or research designs or study different educational contexts to complement general knowledge of the topic.

In terms of the employed methods (i.e. artefact analysis and programming process analysis), an incentive for a methodical instrument validation process surfaces to calibrate the frameworks to measure students' CT more accurately (see Tang et al., 2020). Although the analysis frameworks are founded theoretically upon existing frameworks, the capacity of the methods to operate as credible measures in eliciting the qualities of students' learning could be increased by, for instance, comparing the results of the assessment with other established methods, such as the CT test (Román-González et al., 2017a).

Using the research-designated assessment rubrics for analysing the projects and processes was time-consuming, and the outlined formative assessment system currently carries only hypothetical potential. Enacting formative assessment in practice with the models presented in subsection 6.2.2 would therefore benefit greatly from empirical testing in unique school environments and dynamic classroom situations. Future studies could investigate ways in which teachers and students could utilise the models manually and how (parts of) the assessment could be automatised akin to such tools as Dr. Scratch (Moreno-León et al., 2015) or other real-time dashboard-like assessments (e.g. Koh et al., 2010; 2014; Reppenning et al., 2015).

The assessment developed in this study is intended especially for use in the context of such creative multidisciplinary projects for introductory CT learning at the primary school level as reported by Burke (2012), Hameed et al. (2018) and Whyte et al. (2019). An opportunity for pragmatically valuable research could be further collecting and estimating the 'CT potential' in various kinds of multidisciplinary project templates suitable for processing key substance in primary school curricula (see also Moreno-León & Robles, 2016).

This study examined formative assessment as a collection of rather corporeal pedagogical strategies. The various core tasks involved with it are rich and contextual, though, and the entire process of formative assessment in the classroom could be investigated more focally. For instance, when examining the applicability of the methods of feedback as hypothesised in subsection 6.2.2.3, it may be worthwhile to additionally investigate how students react to the received feedback (e.g. putting effort into reaching learning goals or abandoning them). It would also be meaningful to investigate how students' behaviours link to the teacher's behaviours (e.g. quality of dialogue), what students' beliefs in their capacity to learn are and how the different kinds of feedback (e.g. heavily cued and

rapid) can lead to different kinds of problem-solving strategies (see Black & Wiliam, 1998).

Alongside increasing knowledge about applicable ways to evince student understanding and provide feedback in classroom situations, this study gave reason to produce incipient learning progressions (or ‘criterion sequences’, see Black & Wiliam, 1998) in CT through programming, especially in the context of Scratch (see subsection 6.2.2.1). Fully congruent and more conclusive learning progressions utilising, for instance, the Bloom/SOLO taxonomy (e.g. Meerbaum-Salant et al., 2013) or similar notions as in existing trajectories on specific areas in CT (see e.g. Niemelä, 2018; Rich et al., 2018; 2020; Gane et al., 2021), invite further attention, particularly in terms of CT as viewed through the CEPs.

Additionally, the prospect of aligning the CEPs alongside other disciplinary skills and types of thinking, thus perhaps expediting their way into school curricula as well, can be a major although enticing endeavour. However, as is characteristic of the various definitions and operationalisations of CT, the formulation of the CEPs can develop in future investigations with regard to form, emphasis or level of granularity (e.g. abstracting them into more or less compact modules) for convenience in cross-curricular integration and the design of meaningful learning tasks and instruction (Tang et al., 2020). Yet, with evolving trajectories in CT and increasing knowledge of the higher levels of the competence in particular, learning activities and therefore also the assessment may become more complex. Mapped, rubrics-based assessment frameworks may become impractical, which implies that they may be more relevant for the kind of ‘introductory’, ‘must-know’ or ‘threshold’ substance in CT.

In terms of correcting the limitations of this study, perhaps the most crucial avenue for further research lies in more detailed investigation of collaboration in the context of programming, in particular, ways of (and ways to support) collaborative talk, seeking help and receiving help in this context and information searching. Even though pair programming has been studied – particularly in higher education and in text-based programming environments – more in-depth studies are needed, especially among young learners. Studies could learn to better understand the ways in which social interactions influence school students’ design processes, shared knowledge creation and non-programmatic matters, such as affective factors in collaborative creation, which can also have pivotal effects in learning, especially in creative design contexts. A slightly separate but potentially crucial point of interest may reside in the domain of vocabulary. Language can be particularly central in collaborative problem solving and shared knowledge construction in CT, which includes the verbalisation of potentially unfamiliar and abstract computational matters<sup>23</sup> (Barr & Stephenson, 2011).

---

<sup>23</sup> Earlier research suggests that novice learners tend to think and talk about structural rather than the behavioural or functional parts of the computational system they are designing (Werner et al., 2014). Vandenberg et al. (2020) also found that students’ ability to think about and verbalise abstract computational phenomena can vary according to their prior experience and general developmental level. Piagetian theory of cognitive development could provide a framework for understanding students’ thinking and discourse in CT through programming.



Another crucial avenue for future research lies in a pursuit intrinsic to education – developing students’ cognition. In CT instruction, it is vital for instructors to become aware of the quality of skills and knowledge that students develop while programming. There is no systematic review of empirical studies, such as the ones conducted in this thesis, to synthesise current overall knowledge of how students learn the various CT skills and knowledge through programming. Such a review could result in a coherent overall view of key pedagogical matters at play in CT education; however, conducting such a review may still be somewhat premature. Additional empirical work could still be undertaken to learn to understand students’ thinking at different stages of learning (see e.g. Figure 15) – at the time of naïve skills or knowledge, while becoming acquainted with new conceptual matters, the moments of conceptual change and subsequent moments of putting the new concepts or practices into effect in a self-regulated manner. It remains equally important to learn more lessons with think-aloud methodologies designated to discern students’ in-time thinking while solving computational problems. The same applies for longitudinal studies, which can examine the temporalities in students’ encounters with different CEPs, as can be indicated by artefact analyses and process analyses. Furthermore, important gaps concern the lack of organised empirical knowledge regarding such matters as the effects of feedback (see Figures 16 and 17) to promote conceptual change in different kinds of learning situations.

For future research differing in terms of viewpoints, designs or contexts, this study most essentially prompts the contextualisation of the CEPs in other programming environments or ‘microworlds’, such as with robotic kits (e.g. Barth-Cohen et al., 2018; Chalmers, 2018), physical and unplugged tools (e.g. Brackmann et al., 2019) and digital fabrication contexts (e.g. Iwata et al., 2020; Suero Montero, 2018), in addition to non-programmatic contexts. Professionals working in different fields and specialising in computing may be equipped with valuable understanding concerning what computational methods are used in their respective fields and the ways in which CT enables solving familiar problems better or even illuminates new problems to solve (Denning & Tedre, 2019). The uniform contextualisation of the CEPs, albeit labour-intensive, could lead to added consensus on the topic and ground future scholarly and pragmatic efforts in CT education in similar theoretical premises. It could lead to recognising CT areas that are better learnt in particular contexts or with specific tools or approaches, further strengthening the overall discussion on the inclusion of CT in the curriculum. Additionally, analyses rooted in similar theoretical premises could be utilised in different domains to increase the growing evidence on the transfer of CT from one domain to another.

This study also essentially prompts the design of holistic assessment systems for CT in Scratch, more generally in programming and beyond programming. The assessment systems could be developed to integrate the viewpoints herein and such known assessments as the Bebras challenge (Dagienè & Futschek, 2008) and the CT test (Román-González et al., 2017a). It could also potentially incorporate conceptual and practical notions omitted in this research, such as

'other programming contents' (see Article I), areas of CT that remain outside the CEPs or were difficult to concretise in programming or in Scratch or nuances that the CEPs potentially overlooked. As encouraged in CT assessment (Basso et al., 2018; Grover et al., 2017), the system could assess CT with diagnostic, summative and formative emphases from multiple viewpoints in multiple learning contexts in school and beyond.

Holistic assessment systems should also encompass the more non-cognitive notions in CT to bring the competence closer to one that encompasses 'Knowledge, Skills, Attitudes, Values and Ethical aspects', as theorised by Seiter and Foreman (2013). This study investigated the problem-solving dimension in CT, which can be interpreted to reside alongside the perhaps equally relevant 'computational literacy' dimension (i.e. more social and societal matters of computing) (see Høholt et al., 2021; Lonka et al., 2018; Mertala et al., 2020). Despite nascent empirical research (e.g. Kong et al., 2018; Mannila et al., 2020), this dimension is currently rather ill-defined, abstruse and scarcely studied. This study thus encourages concretising what are the more cross-contextual computational attitudes, perceptions or dispositions – stemming potentially from societal reality and adapted to the core functions of primary education – that students could or should gain in primary education. Future research could proceed to investigate how such dispositions could be incorporated in programming education or more general-level CT education in, for instance, the form of developing learning activities that include interpreting and/or designing computer programs and other ways of being engaged with the computational world. A natural next step could also be related to the assessment of such dispositions.

## 6.5 Closing remarks

CT is a contended term that has received much attention in educational discourse in recent decades. It is vital to appreciate that it is an age-old idea and is only new and understandably more acute in today's world in relation to the rapid development and ubiquity of digital technology. Nonetheless, educational scholars and practitioners have begun to actively ask such questions as 'What is CT exactly?', 'Who does it belong to?' and 'How much do students need it?' Straightforward answers have been difficult to provide because the topic is so young, unsettled and directed to the future, which is difficult to predict very accurately.

The topic has been put into practice and developed actively in schools, libraries, science centres, non-profit organisations and after-school clubs through such initiatives as robotics, making and digital fabrication, even for years before CT jumped in front of the public eye. Scientific research has also only just come into play to strengthen a research-based understanding of the term. A concurrent richness and difficulty has been that it has brought together experts from different worlds, such as crafts, engineering, higher education, basic education, early childhood education, computing, CS and information technology. Discussions

have in many ways been ground-breaking but also cumbersome due to the lack of many shared qualities, such as vocabularies, disciplinary paradigms and perhaps even interests.

CT is still a highly problematic term for several reasons. First, the terminology, which would greatly enable discussing the topic by referring to the same ideas, is still not well-established. Second, the educational goal of CT can be viewed in different ways; CT can allegedly foster generic procedural skills, provide computational problem-solving skills, teach mere coding or enrich a kind of computational literacy. Consequently, it can be either a very rigid disciplinary set of methods that must be explicitly taught or more akin to a pervasive cognitive foundation that stealthily transcends entire curricula through such skills as decision making and decomposition. Although some goals can be justified as too limited, too hopeful, too ambitious, too industry-centred or too vague, all of them can be relevant in some ways. How can such a slippery thing be grabbed and framed conveniently?

That said, one of the less studied albeit highly compelling goals of CT deserves a distinct mention. The importance of teaching students computational literacy skills has been rightfully promoted in recent years. It is becoming inevitably important to teach young people to understand what kinds of computational phenomena shape the digital world, how they influence our lives and how we should learn to comprehend them and relate to them as critical agents of the world. There is much popular talk about recommender algorithms, Internet privacy, information bubbles, storing and selling personal information, digital warfare and online tracking, among others, each of which have programmed things and CT behind them. It seems crucial to open the field from the viewpoint of educational sociology. An immediately following question could be whether something akin to computational literacy should be coined as a separate term or just be positioned under 'regular' media literacy or multiliteracy. Nonetheless, the importance of CT literacy seems all the more significant in the ever-accelerating era of digitalisation and robotisation that drives a bigger and bigger gap between school and the real world.

Perhaps nascent solutions are in sight. Perhaps computational literacy (an understanding of computing and the computational world) is crucial for everyone and computational problem solving (skills to apply and create with computational technologies) is useful for 'some' people. Currently, it feels safe to only say that those 'some' are 'those who will do CT jobs in the future'. In other words, we need insight with respect to how the core principles of CT are utilised or could be utilised in different types of problem-solving situations (e.g. in different sectors of work) in the real world. It is not likely that all professional workers need 'creative CT' yet, although the computationalisation of different work sectors is evident. Still, that does not necessarily mean there should be a divorce between engineers who build, coders who code and other workers who do the 'normal' work. It seems more appropriate to think that everyone needs a basic understanding (e.g. vocabulary, knowledge of the methods and techniques) of compu-

tational problem solving to be able to communicate ideas with software designers equipped with deeper CT. 'How can CT assist me in my work?' Perhaps answers to questions such as this could also facilitate the inclusion of CT more expansively, and particularly in non-programmatic situations, across different subjects in school curricula.

Overall, in primary education CT can be viewed as a basic but multifaceted cornerstone of 'computational knowing and doing', which provides a springboard for gaining deeper skills to solve complex problems in real-life situations in the future. More prophetic visions even portray it as an entirely new kind of competence, something that rises to the same level as writing and calculating, that enables conjuring unseen things to make the world a better place. Although such claims may seem deranged, CT has already brought us the Internet, social media, smart phones and mind-boggling applications of artificial intelligence.

As was perhaps expected in the research and development of this complex and unestablished competence, no easy and all-encompassing model for teaching and learning emerged. In contrast, this study showed that several matters can and perhaps need to be taken into account when attempting to teach, learn, assess and study CT comprehensively, systematically and reliably. Perhaps it is therefore sensible to in a way return to the starting point of this thesis and interpret CT more holistically rather than too atomistically to avoid overly detailed and too broken approaches ('not seeing the forest for the trees'). With that in mind, a guideline for educational practitioners to do 'at least something related to CT' can be a good starting point to introduce students and practitioners themselves to the world of computing at this starting point of its hopefully enduring and gradually fortifying voyage to formal education.

The author strongly suggests that the arguments in this thesis stress an invitation to put as much effort as possible into continuing to develop this topic educationally. The results of this thesis have already begun to be put into place in national curricular consultation, in-service and pre-service teacher training and the design of international learning materials, but there is much work yet to be done. The efforts of this study are aimed at a broad pragmatic interest of educational research—developing the school. There are several known challenges with CT amidst this interest, and they can be seen to concern such key elements as learning and learning environments, teacher knowledge and collaboration, shared leadership and teamwork and school partnerships (Korhonen & Lavonen, 2016). Next to even the most finely elucidated competences, laid out learning trajectories, or fine-tuned pedagogical models, researchers need to better understand key practical requisites faced by learners, teachers, school leaders and educational policy makers alike in designing and enacting CT education. Amidst the ongoing struggles, the big question remains: How can educational research guide change in the school? What key questions remain unanswered regarding, for example, learner dispositions, teachers' understanding of CT and their own beliefs about developing their CTPACK, leadership and curricular work?

At the end of this study, the author would like to state his personal professional wish that all students would have the opportunity to gain an interest in

programming and cultivate strong basic CT skills in order to have a good chance of deepening them in their further studies, work life and other life situations. That said, it is desirable that the role of programming would not diminish in future core curricula or be left to an elective status, as has unfortunately happened in the core curriculum of the Finnish upper secondary school. Instead, each student's right to study CT should continue unimpeded from pre-school to higher education. Meanwhile, it is highly desirable that vocational education and the different scientific disciplines at universities embrace the adventure of seeking the possibilities that CT can provide for the problem-solving methodologies characteristic of their fields.

How the learning of the new and multifaceted CT can be best facilitated and supported still remains partially shrouded, but despite the challenges encountered, the current direction of ongoing developments looks promising. CT is studied to an increasing extent, programming belongs to everyone in the Finnish primary school core curriculum and teachers seem to increasingly be doing something about it. Still, the growth of CT in schools has been surprisingly slow, and the topic does not seem to grab the interest of too many educational professionals, unlike other technological topics such as those aimed at supporting learning processes in general or providing immersive learning experiences (e.g. learning with virtual reality applications). Digital technology and computing as targets of learning rather than facilitators of learning are increasingly important, but the demand and supply do not seem to meet. Nonetheless, pedagogical models are being constantly developed for different educational situations, and the challenges of adopting CT and programming in schools and other educational institutions are being constantly studied nationally and internationally.

Meanwhile, hundreds of motivated and excited students compete annually in programmatic events, such as sumo wrestling with Lego robots and game making with Scratch (see Figure 18), which only demonstrates that the topic can generate a great deal of interest and excitement very broadly. Based on these facts, we have come far from approximately 70 years ago when, according to Denning and Tedre (2019), programming was viewed mainly as a 'black art' that was accessible mainly to the elite.



Figure 18. Scratch programming during GameDev, a game design competition organised at the Innokas programming and robotics tournament (photo: Miika Miinin)

## YHTEENVETO

Tietokonevallankumous on täällä! Merkittävä osa nykyisin käyttämistämme monimutkaisista laitteista, kuten älypuhelimet ja autot, ja palveluista, kuten pankki- ja viihdepalvelut, ovat pohjimmiltaan tietokoneita ja niiden suorittamia tietokoneohjelmia. Siitä huolimatta edistyneemmätkin tietokoneet ovat vain tietojenkäsittelylaitteita, jotka noudattavat täysin samoja käsitteellisiä periaatteita kuin vaikkapa alkeelliset helmitaulut. Vasta 1900-luvulla tietokoneiden nykypäivänä toteuttamia tiedonkäsittelytapoja sähköistettiin ja automatisoitiin, tietojenkäsittelytiedettä virallistettiin ja digitaalisia tietojenkäsittelykoneita rakennettiin ja tuotiin koteihin ja työpaikoille. Erilaisten tietokoneohjelmien luonti on sittemmin tuottanut valtavia hyötyjä, kuten lääketieteen uudistuksia ja lentoturvallisuuden simulaatioita, mutta toki myös vakavia huolenaiheita, kuten verkkovalvontaa ja informaationsodankäyntiä.

Samaan aikaan tietokoneiden suorittamien ohjelmien itse tekeminen eli ohjelmointi on palannut kouluihin pakollisena tai valinnaisena oppisisältönä eri maissa, mukaan lukien Suomessa. Erilaiset lapsille ja nuorille tarkoitetut pelin- ja leikinomaiset ohjelmoinnin kontekstit, kuten robotiikka, pelinteko ja ”värkkäily” ovat runsastuneet merkittävästi. Erityisen suosittujen ympäristöjen joukossa on tässäkin tutkimuksessa tarkasteltu ilmainen, verkkoselaimella käytettävä ohjelmointiympäristö Scratch, jolla voi luoda monipuolisesti erilaisia interaktiivisia tarinoita, animaatioita ja pelejä ponnistaen omasta luovuudesta ja omista kiinnostuksenkohteista. Omiin Scratch-ohjelmointitöihin suunnitellaan erilaisia digitaalisia hahmoja, kuten vaikkapa taruolentoja, sekä tapahtumapaikkoja, joille ohjelmoidaan erilaisia toimintoja, kuten liikettä, puhetta ja ääntä liitettämällä yhteen kuvakepohjaisia koodilohkoja.

Ohjelmoinnin oppimisen erääksi keskeiseksi tavoitteeksi koulussa on esitetty *ohjelmoinnillisen ajattelun* oppiminen. Ohjelmoinnillinen ajattelu voidaan tulkita monitahoisena tieto- ja taitokokonaisuutena, joka ammentaa tietojenkäsittelytieteen tieteenalalle ominaisista ajattelun ja tekemisen tavoista. Sen tarpeellisuuden perusteiksi on esitetty muun muassa lisääntynyt ammattikoodaajien tarve, ohjelmoinnillisen mallien ja menetelmien yleistyminen eri työaloilla, yleismaailmallisten tietojen ja taitojen oppimisen mahdollisuudet sekä ymmärryksen jalostaminen alati ohjelmoinnistuvasta yhteiskunnasta.

Ohjelmoinnillinen ajattelu on kuitenkin hyvin uusi käsite, jota koskeva tutkimus ja tietämys on vasta hiljalleen kasvamassa. Ohjelmoinnin ja sitä kautta opittavan ohjelmoinnillisen ajattelun opetusta on jalkautettu eri koulutusasteille ja eri maihin erilaisten kasvatustavoitteiden linjassa erilaisin tavoin ja kenties erilaisin odotuksinkin. Tuoreiden tutkimusten mukaan erityisesti suomalaisopettajat eivät koe kykenevänsä opettaa ohjelmointia, ja he painottavat ohjelmoinnillisen ajattelun oppimista omassa opetuksessaan verrattain vähän. Suomalaisoppilaiden ohjelmoinnin osaaminen on tutkitusti matalaa, ja ohjelmointi tiedetään muutenkin hyvin haasteelliseksi aktiviteetiksi erityisesti taidon oppimisen alkuvaiheissa.

Tässä väitöstutkimuksessa tutkittiin ohjelmoinnillisen ajattelun oppimisen tukemista Scratch-ohjelmoinnin kontekstissa peruskoulun luokkahuonetilanteissa. Tutkimuksessa toteutettiin neljä toimenpidettä:

- (1) ohjelmoinnillisen ajattelun kasvatustavoitteiden täsmentäminen ohjelmoinnin kontekstissa,
- (2) ohjelmoinnillisen ajattelun arviointimenetelmien tarkastelu Scratchissa,
- (3) uusien arviointimenetelmien kehittäminen Scratch-ohjelmointitilanteisiin ja
- (4) rikkaan empiirisen tiedon hankinta 4.-luokkalaisoppilaiden ohjelmoinnillisesta ajattelusta Scratchissa.

Tutkimuksen toteuttamisen pohjana toimi taustakirjallisuudesta uudella tavalla muotoiltu *ohjelmoinnillisen ajattelun opetuksen perusprinsipiistä* koostuva teoreettinen viitekehys. Perusprinsipiellä kuvattiin konkreettisesti, millaista ymmärrystä oppilaat voivat tarkalleen ottaen omaksua esimerkiksi erilaisten algoritmien luomisesta ja tietokoneiden ymmärtämisen eli datan hyödyntämisestä ohjelmoinnissa. Viitekehyksen mukaan ohjelmoinnillisen ajattelun opetuksen päämääränä on ruokkia ymmärrystä siitä, mitä tietojenkäsittelyllä voidaan tehdä, miten tietokoneet käsittelevät tietoa sekä miten tietojenkäsittelyn erilaisia työkaluja, malleja ja ideoita voidaan käyttää ratkaisemaan erilaisia oikean elämän ongelmia. Sen mukaan ohjelmoinnillinen ajattelu voidaan tulkita eri tilanteissa sovellettavana ylätasoa osaamisena, ja ohjelmointi on puolestaan alemman tason osaamista, joka voi tietyillä tavoilla saada aikaan ohjelmoinnillisen ajattelun oppimista.

Tutkimuksen pedagogisena kehysteorianä toimi konstruktionistinen oppimiskäsitys. Konstruktionistiseen oppimiseen kuuluu ajatus siitä, että erilaisten konkreettisten tuotosten (esim. ohjelmointitöiden) luominen ulkoistaa oppijan oman ajattelua. Tosielämän ilmiöiden opettaminen tietokoneelle ohjelmoiden näin ollen mahdollistaa ja osoittaa kyseisen ilmiön oppimista. Konstruktionismia on sovellettu usein erityisesti löytämällä oppimisen yhteydessä: oppiminen tapahtuu spontaaneissa tilanteissa, joissa uutta tietoa tarvitaan yllättäen jonkin käytännön ongelman ratkaisemiseksi. Tutkimuksessa kuitenkin tulkittiin, että vaikka spontaani löytäminen olisikin oppimisen peruserä, oppimista voi olla toisinaan tarpeen tukea ja ohjata. Keskeisiä toimia voivat olla erityisesti formatiivisessa arvioinnissa määritellyt tukistrategiat: selkeiden oppimistavoitteiden asettaminen, oppilaiden osaamisen määrätietoinen selvittäminen ja tarkoituksenmukainen palautteenanto. Tutkimuksessa tulkittiin, että näitä strategioita voidaan mahdollistaa oppilaiden autenttisissa ohjelmoinnillisen ajattelun oppimisen tilanteissa tarkastelemalla oppilaiden Scratch-ohjelmointitöitä ja heidän ohjelmointikäytänteitään Scratchissa.

Tutkimus oli luonteeltaan monimenetelmäinen, ja se toteutettiin kolmen tieteellisen vertaisarvioidun tutkimusartikkelin kautta. Artikkelissa I toteutettiin systemaattinen kirjallisuuskatsaus (30 kirjallisuuslähdettä), joka kokosi erilaisia tapoja arvioida ohjelmoinnillisen ajattelun perusprinsiipien oppimista Scratchissa. Toisin sanoen katsauksessa koostettiin laajasti erilaisia tapoja arvioida



Scratch-ohjelmoinnin sisältöjä ("mitä" oppilaat ohjelmoivat) ja ohjelmointikäytänteitä ("miten" oppilaat ohjelmoivat). Näitä sisältöjä ja käytänteitä tulkittiin samalla uudella tavalla juurikin ohjelmoinnillista ajattelua ruokkivina sisältöinä ja käytänteinä.

Artikkelien II ja III empiiriset tapaustutkimukset kokosivat monipuolista aineistoa oppilaiden ohjelmoinnillisesta ajattelusta käytännön luokkahuonetilanteissa. Tutkimukseen kerättiin aineistoa 57:ltä 4.-luokan oppilaalta (62% tyttöjä, 38% poikia) kolmesta keskikokoisen keskisuomalaisen koulun rinnakkaisluokasta. Oppilaat osallistuivat 12 oppitunnin mittaiseen Scratch-ohjelmoinnin opintojaksoon vuonna 2017. Oppilaat olivat tutkimuksen alettua kokemattomia ohjelmoijia, joten jakson tavoitteena oli tutustuttaa oppilaat Scratch-ohjelmoinnin peruskäyttöön ja ohjelmoinnillisen ajattelun perusteisiin. Oppilaat työskentelivät jakson aikana pääsääntöisesti pareittain saman tietokoneen äärellä ohjelmoiden yhteisiä luovia ohjelmointitöitään.

Tutkimuksessa kerättiin kaksi aineistokokoelmaa: oppilaiden tekemät Scratch-ohjelmointityöt (N=325) sekä videotallenteet neljän oppilasparin ohjelmointityöskentelystä jakson viimeisten vapaavalintaisten ohjelmointitöiden valmistamisesta kahden oppitunnin ajalta. Molemmissa tutkimusartikkeleista kehitettiin ja sovellettiin uusia analyysimenetelmiä ohjelmoinnillisen ajattelun teoreettisesti perusteltuun arviointiin.

Artikkelissa II oppilaiden tekemistä Scratch-ohjelmointitöistä tarkasteltiin erityisesti ohjelmoituja "koodauskaavoja" eli yleistason ohjelmointitekniisiä malleja, joilla ohjelmointitöiden toiminnallisuuksia oli ohjelmoitu, sekä ennen kaikkea näiden sisältöjen viitteitä ohjelmoinnillisen ajattelun oppimiseen. Tutkimuksen mukaan oppilaat olivat laajasti joskaan eivät täysin tyhjentävästi tekemisissä ohjelmoinnillisen ajattelun perusprinsiipien kanssa opintojakson aikana. Tulokset antoivat ymmärtää, että oppilaat saattoivat oppia ohjelmoinnillisen ajattelun perusprinsiipejä vain osittain eikä kaikkia kovin syvällisesti. Oppiminen näytti tapahtuvan luontaisemmin tietynlaisten ohjelmointitekniisten mallien kautta, jotka ovat tyypillisiä Scratchissa toteutettaville harjoitustöille eli yleisimmiten animaatioille ja tarinallisille töille. Ohjelmoinnillisen ajattelun laaja-alaisempi oppiminen näytti edellyttävän päämäärätietoisempia opetuksellisia valintoja, kuten ajoittaista tiedon suoraa esittämistä pelkän "etsimisen ja löytämisen" sijaan, aiemmin käsiteltyjen ohjelmoinnillisten mallien käytön osaamisen vahvistamista sekä ennen kaikkea monimutkaisempien pelinomaisten ohjelmointitöiden toteuttamista järjestettyjen oppituntien lisäksi.

Artikkelissa III oppilaiden pariohjelmointiprosesseista tarkasteltiin erilaisia tapoja, joilla parit toteuttivat yhdessä ohjelmoinnilliseen ajatteluun kuuluvia käytänteitä, kuten luovaa suunnittelua, yhteistoiminnallista työskentelyä ja "debuggaamista". Artikkelin päähavaintoihin kuului löydös siitä, että luovien ohjelmointitöiden alkusuunnittelu on tärkeää niin ohjelmointiprosesseissa oleellisesti hämmäyttävien sudenkuoppien välttämiseksi kuin toisaalta uusien oppimismahdollisuuksien avaamiseksi. Toisaalta omien ohjelmointitöiden spontaani kehittäminen ja uusien ideoiden löytäminen matkan varrella voivat johdatella ohjelmointiprosessia uutta oppimista avaaviin suuntiin, jolloin vaatimuksena voi toisaalta

olla ohjelmoinnillisesti taitavan opettajan tiivis tuki uusien asioiden oppimisessa. Tulokset osoittivat myös, että vaikka oppilaiden itsenäinen pariohjelmointityöskentely voi olla monilta osin tehokasta ja tuottoisaa, itsenäisen työskentelyn sujuvuus ja ajoittaiset avuntarpeet voivat vaihdella yksilöllisesti. Oppilaiden omajohtoisen työskentelyn aikana voi nousta erityisesti pedagogisesti huomionarvoisia ei-toivottuja toimintatapoja niin ohjelmoinnillisten virheiden korjaamisessa, sopivien ohjelmoinnillisten sisältöjen (esim. koodirakenteiden) löytämisessä ja käyttämisessä kuin yhteisen tiedonrakentelun osalta sekä parin sisäisesti että ulkopuolisten tietoresurssien, kuten luokkahuonetovereiden hyödyntämisessä. Löydökset korostavat kaikkiaan, mitkä ohjelmoinnin ja oppimisen ohjaamisen osa-alueet ovat kenties keskeisimpiä sekä miten oppilaskeskeistä ongelmanratkaisua prosessia voitaisiin luokkahuoneessa tukea.

Sekä artikkelin I kirjallisuuskatsaus että artikkelien II ja III empiiriset löydökset antavat kokonaisuudessaan viitteitä ohjelmoinnillisen ajattelun oppimiseen sekä valtakunnallisen perusopetuksen opetussuunnitelman että ruohonjuuritason opetuksen näkökulmista.

Opetussuunnitelman näkökulmasta on kaikkiaan tärkeää tunnistaa ohjelmoinnillisen ajattelun osaamisen moninaiset hyödyt ja siihen kuuluvat moninaiset tietämisen ja taitamisen tavat, joita voidaan harjoitella eri tavoin esimerkiksi ohjelmoimalla. Vaikka tutkimuksessa ei varsinaisesti arvioitu ohjelmoinnin roolia opetussuunnitelmassa, voidaan kevyelläkin vertailulla todeta ohjelmoinnillisen ajattelun opetuksen perusprinsiipien esiintyvän nykyisessä peruskoulun opetussuunnitelmassa kapea-alaisesti. Ohjelmoinnillista ajattelua saatetaan toki oppia eri oppiaineissa ilman erillismainintaakin, mutta kyseisen osaamiskokonaisuuden ainutlaatuinen, tietojenkäsittelyn alueesta ammentava luonne voi edellyttää sen opettamista ja oppimista harkituin menetelmin esimerkiksi sisällyttämällä tässä tutkimuksessa konkretisoituja prinssiipejä opetussuunnitelman eri alueille. Käytännössä Scratchin kaltaisilla ohjelmointiympäristöillä voidaan oppia ohjelmoinnillista ajattelua ”pienin annoksin”, mutta on erityisen tärkeää tulkita ohjelmoinnillinen ajattelu pitkäkestoisesti ruokittavana, kokonaisvaltaisena osaamisena. Sitä ei välttämättä opita parhaiten tarkasti eriteltyjen alakriteerien kautta, vaan kokonaisvaltaisesti teknologiaa hyödyntävän tietojenkäsittelyä hyödyntävän ongelmanratkaisun kautta erilaisissa oppimistilanteissa ja erilaisilla työvälineillä. Karkeasti ohjelmoinnillisen ajattelun oppimäärä voidaan raimittaa koulun näkökulmasta ”jokaisen työkaluksi, joidenkin ammattityökaluksi.”

Ruohonjuuritason kannalta tutkimus tuotti konkreettisia keinoja oppilaiden ohjelmoinnillisen ajattelun oppimisen tukemiseen ohjelmoinnin kautta. Näitä keinoja voidaan eritellä kolmen keskeisen pedagogisen strategian kautta: (1) oppimistavoitteiden asettaminen, (2) oppilaiden osaamisen selvittäminen ja (3) palautteenanto.

(1) Oppimistavoitteiden asettaminen. Tutkimuksen tuloksina syntyneet monipuoliset kokoelmat ohjelmoinnillisen ajattelun perusprinsiipejä ja niiden kontekstualisoimia Scratch-ohjelmointisisältöjä ja -käytänteitä voidaan soveltaa oppimistavoitteina kouluopetuksessa erilaisten luovien ja monialaisten Scratch-

ohjelmointitöiden teossa. Tulokset antoivat lisäksi perusteita ohjelmointisisältöjen järjestämiseen karkeina oppimisen etenemisen polkuina Scratchissa (kuviot 13 ja 14). Tiivistäen: ohjelmoinnillisen ajattelun oppiminen voidaan aloittaa alkeista laatimalla interaktiivisia animaatioita ja tarinoita, joissa sovelletaan oletettavasti perustavanlaatuisempia ohjelmoitavia ohjelmointitekniisiä malleja. Oppimisen myöhemmissä vaiheissa jatketaan toteuttamaan monimutkaisempia pelinomaisia ohjelmointitöitä, joissa ohjelmoinnilliset mallit ja ratkaisut ovat karkeasti ottaen edistyneempiä ja monimutkaisempia. Oppilaat voivat edetä polulla ja seurata omaa osaamistaan esimerkiksi sen mukaan, kuinka hyvin he kokevat osaavansa käyttää, muokata tai luoda ymmärrettävyydeltään eritasoisia malleja ja ratkaisuja. Oppilaiden on tärkeä myös oppia ymmärtämään tilannekohtaisesti, millaiset ohjelmointikäytänteet voivat viedä omaa ohjelmointityötä ja oppimista eteenpäin.

(2) Oppilaiden osaamisen selvittäminen. Oppilaiden kulloistenkin osaamistasojen selvittämisen tavoitteena on valjastaa saatu tieto oppimisen yksilölliseksi edistämiseksi. Oppilaiden ohjelmointitöihinsä ohjelmoimia malleja ja ratkaisuja sekä heidän toteuttamia ohjelmointikäytänteitä voidaan tulkita osaamisen osoituksina: millaista käsitteellistä ymmärrystä (esim. erilaisista algoritmeista) ja praktista osaamista (esim. debuggauksesta) he ovat hankkineet ja pystyvät soveltamaan käytännössä. Osaamisen taso voi näyttäytyä eri tavoin, kuten esimerkiksi oikein ohjelmoitujen ratkaisujen runsaan määrän tai toistuvan teknisesti onnistuneen käytön kautta. Yleisesti ottaen osaamisen kriteerinä voidaan pitää ohjelmoinnillisten ratkaisujen ja ohjelmointikäytänteiden hyötyjen ja käyttötapojen ymmärtämistä ja niiden soveltamista omiin ohjelmointitöihin koodilohkoina tai tehokkaina ohjelmoinnin tapoina tilannekohtaisesti tavoitteellisella tavalla. Osaamisen todentaminen voi edellyttää pitkäkestoista tai syvällistä arviointia, ja koko ohjelmoinnillisen ajattelun kokonaisuuden arviointi voikin olla haastava tehtävä. Sen sijaan tutkimuksen tulokset vihjasivat, että arviointi voi olla hedelmällisintä sellaisten ohjelmointitekniisten mallien ja ohjelmointikäytänteiden osalta, joita oppilaat eivät näytä omaksuvan kovin itseohjautuvasti tai helposti (ks. taulukko 4).

(3) Palautteenanto. Oppilaille annettavan palautteen tarkoituksena on ohjata oppimista sen nykytilasta tavoiteltavaan tilaan eli oppimistavoitteita kohti. Vaikka ohjelmoinnin oppimiseen on konstruktionistisen oppimiskäsityksen mukaan tarkoitus jättää tilaa itseohjautuvalle oppimiselle esimerkiksi tietokoneen antaman jatkuvan palautteen (mm. ohjelmointivirheiden eli bugien) ansiosta, oppilaat näyttävät tarvitsevan toisinaan tukea tietokoneen välittämän tiedon ymmärtämisessä. Tärkeä tapa mahdollistaa jatkuvan palautteen saaminen voi olla rohkaista sosiaalista tiedonrakentelua niin pariohjelmoinnin, vertaispalautteen kuin esimerkiksi Internet-tiedonhakujen kautta. Näiden käytänteiden toteuttaminen voi olla kuitenkin oppilaille omatoimisesti haasteellista, jolloin opettajien voi olla tarpeen tukea oppimista niin ennen aikaisesti (esim. havainnollistamalla oppimisen polkua) kuin spontaanisti esiin tulevissa tilanteissa. Spontaanin palautteen muodot väistämättä vaihtelevat, sillä ohjelmoinnillisessa ajattelussa ollaan tekemisissä luonteeltaan hyvinkin vaihtelevien oppisisältöjen kanssa, kun

taas myös oppilaiden osaamistasot ja avuntarpeet sekä sopivat yksilöllisen tuen muodot voivat vaihdella. Tukea voidaan tarvita karkeasti ottaen niin esiin tulevien ongelmien ratkaisemiseen (esim. debuggauksessa) kuin uusien oppimisen väylien avaamiseenkin (esim. tehokkaampien ohjelmointitekniikoiden löytämisessä). Epäsuorempi apu voi rohkaista itsenäisempään ongelmanratkaisuun, mutta joskus suurempikin apu voi olla tarpeen. Tutkimuksen tulokset konkretisoivatkin erilaisia spontaanin tuen muotoja (kuvio 16), joiden edellytyksenä voi kuitenkin olla opettajan osaaminen ohjelmoinnillisessa ajattelussa tai muunlaiset puitteet, kuten riittävä aika yhteiselle tiedonhauille, kokeneempien oppilaiden vertaistuki tai automaattiset arviointityökalut.

Tutkimus herätti lukuisia jatkotutkimusaiheita. Ensinnäkin tutkimuksessa sovellettujen arviointimenetelmien tieteellinen validointi vahvistaisi niiden kykyä mitata oppilaiden osaamista. Lisäksi tutkimuksessa kehitettyjen, yllä kuvattujen oppimisen tukemisen mallitoteuttamistapojen kokeileminen dynaamisissa luokkahuonetilanteissa toisi lisätietoa niiden toteuttamisen soveltuvuudesta käytännössä. Formatiivinen arviointi onkin monimuotoinen prosessi, jonka tiimoilta on syytä tutkia myös esimerkiksi opettajan ja oppilaan vuorovaikutusta sekä sitä, kuinka oppilaat reagoivat saamaansa palautteeseen. Lisäksi ohjelmoinnillisen ajattelun perusprinsiippien järjestäminen väkevämmiksi oppimispoluiksi, niiden rinnastaminen erilaisiin tietämisen ja taitamisen tapoihin (esim. matemaattisiin taitoihin ja luovuuteen) sekä niiden oppiminen erilaisten monialaisten ohjelmointityömallipohjien kautta voisi edistää ohjelmoinnillisen ajattelun jalkautumista opetussuunnitelmaan.

Myös tutkimuksessa esiin tulleet vajeet jättivät osaltaan tärkeitä tutkimusaiheita jatkoon. Tärkeää olisi ennen kaikkea tutkia koululaisten yhteistoiminnallisuutta ohjelmoinnissa erityisesti yhteistoiminnallisen puheen, vertaisoppimisen, opettajan tuen ja tiedonhaun näkökulmista. Yhtäältä tärkeää on syventää ymmärrystä siitä, kuinka oppilaat tarkalleen ottaen ajattelevat ja kuinka heidän ajattelunsa muuttuu ohjelmoinnillisen ajattelun oppimisen eri vaiheissa.

Tämän nuoren ja vakiintumattoman aihepiirin tutkimus rohkaisi myös tutkimaan ohjelmoinnillisen ajattelun perusprinsiippien opettamista ja oppimista muissa ohjelmointiympäristöissä ja myös ei-ohjelmoinnillisissa tilanteissa. Tutkimus on tärkeää yhteisymmärryksen lisäämiseksi sekä esimerkiksi ohjelmoinnillisen ajattelun oppimisen siirtovaikutuksen tutkimuksessa. Kaikkiaan tutkimus rohkaisee jatkotutkimuksia jalostamaan kokonaisvaltaisia arviointimalleja, joissa huomioidaan ohjelmoinnillisen ongelmanratkaisuosaamisen lisäksi myös oppilaiden ”ohjelmoinnillisen medialukutaidon” ulottuvuus. Aihepiiriä koskevassa tutkimuksessa on kaikkiaan ensisijaisen tärkeää pitää yllä ymmärrystä siitä, millaiset seikat oppimisessa, opettajuudessa, koulun johtajuudessa ja koulun yhteistyöverkostoissa edistävät ja estävät tämän uuden mutta yhteiskunnallisesti kasvavan tärkeän aihepiirin jalkautumista koulun arkeen.

## REFERENCES

- Ackermann, E. (2001). *Piaget's constructivism, Papert's constructionism: What's the difference?* MIT Media Laboratory.  
[https://learning.media.mit.edu/content/publications/EA.Piaget%20\\_%20Papert.pdf](https://learning.media.mit.edu/content/publications/EA.Piaget%20_%20Papert.pdf)
- The Association for Computing Machinery (2005). *Computing curricula 2005*. ACM.  
<https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2005-march06final.pdf>
- Aho, A. (2011). Computation and computational thinking. *Ubiquity*, 2011 (January), Article No. 1. <https://doi.org/10.1093/comjnl/bxs074>
- Ally, M., Darroch, F., & Toleman, M. (2005). A framework for understanding the factors influencing pair programming success. In H. Baumeister, M. Marchesi & M. Holcombe (Eds.), *Extreme programming and agile processes in software engineering. XP 2005. Lecture notes in computer science, vol 3556* (pp. 82-91). Springer. [https://doi.org/10.1007/11499053\\_10](https://doi.org/10.1007/11499053_10)
- 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 & Society*, 19(3), 47-57. <http://www.jstor.org/stable/jeductechsoci.19.3.47>
- Arisholm, E., Gallis, H., Dybå, T., & Sjøberg, D. I. K. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2), 65-86.  
<https://doi.org/10.1109/TSE.2007.17>
- Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From Scratch to "real" programming. *ACM Transactions on Computing Education*, 14(4), Article No. 25. <https://doi.org/10.1145/2677087>
- Armoni, M. (2019). Computing in schools: On the knowledge of CS teachers' educators. *ACM Inroads*, 10(2), 10-13. <https://doi.org/10.1145/3324885>
- Balanskat, A., & Engelhardt, K. (2015). *Computing our future. Computer programming and coding - Priorities, school curricula and initiatives across Europe*. European Schoolnet.  
<http://www.eun.org/resources/detail?publicationID=661>
- Balanskat, A., Engelhardt, K., & Ferrari, A. (2017). The integration of computational thinking (CT) across school curricula in Europe. *European Schoolnet Perspective, Issue no. 2, April 2017*. European Schoolnet.
- 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.  
<https://doi.org/10.1145/1929887.1929905>
- Basso, D., Fronza, I., Colombi, A., & Pahl, C. (2018) Improving assessment of computational thinking through a comprehensive framework. In M. Joy & P. Ihantola (Eds.), *Proceedings of the 18th Koli calling international conference*

- on computing education research (Koli calling '18) (Article No. 15). ACM. <https://doi.org/10.1145/3279720.3279735>
- Basu, S., Kinnebrew, J. S., & Biswas, G. (2014). Assessing student performance in a computational-thinking based science learning environment. In S. Trausan-Matu, K. E. Boyer, M. Crosby & K. Panourgia (Eds.), *Intelligent tutoring systems. ITS 2014. Lecture notes in computer science, vol 8474*. Springer. [https://doi.org/10.1007/978-3-319-07221-0\\_59](https://doi.org/10.1007/978-3-319-07221-0_59)
- Basu, S. (2019). Using rubrics integrating design and coding to assess middle school students' open-ended block-based programming projects. In E. K. Hawthorne & M. A. Pérez-Quiñones (Eds.), *Proceedings of the 50th ACM technical symposium on computer science education (SIGCSE '19)* (pp. 1211–1217). ACM. <https://doi.org/10.1145/3287324.3287412>
- Ben-Ari, M. (1998). Constructivism in computer science education. In J. Lewis & J. Prey (Eds.), *Proceedings of the 29th SIGCSE technical symposium on computer science education (SIGCSE '98)* (pp. 257–261). ACM. <https://doi.org/10.1145/274790.274308>
- Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education, 72*, 145–157. <https://doi.org/10.1016/j.compedu.2013.10.020>
- Bers, M. U., González-González, C., & Belén Armas Torres, M. (2019). Coding as a playground: Promoting positive learning experiences in childhood classrooms. *Computers & Education, 138*, 130–145. <https://doi.org/10.1016/j.compedu.2019.04.013>
- Binkley, M., Erstad, O., Herman, J., Raizen, S., Ripley, M., Miller-Ricci, M., & Rumble, M. (2012). Defining twenty-first century skills. In P. Griffin, B. McGaw & E. Care (Eds.), *Assessment and teaching of 21st century skills* (pp. 17–66). Springer. [https://doi.org/10.1007/978-94-007-2324-5\\_2](https://doi.org/10.1007/978-94-007-2324-5_2)
- Black, D., & Wiliam, D. (1998). Assessment and classroom learning. *Assessment in Education: Principles, Policy & Practice, 5*(1), 7–74. <https://doi.org/10.1080/0969595980050102>
- Black, D., & Wiliam, D. (2009). Developing the theory of formative assessment. *Educational Assessment, Evaluation and Accountability, 21*(1), 5–31. <https://doi.org/10.1007/s11092-008-9068-5>
- Blikstein, P. (2020). Cheesemaking emancipation: A critical theory of cultural making. In N. Holbert, M. Berland & Y. Kafai (Eds.), *designing constructionist futures: The art, theory, and practice of learning designs* (pp. 115–126). MIT Press.
- Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., & Koller, D. (2014). Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences, 23*(4), 561–599. <https://doi.org/10.1080/10508406.2014.954750>
- Bloom, B. S. (1956). *Taxonomy of educational objectives. Vol. 1: Cognitive domain*. McKay.

- Bocconi, S., Chiocciariello, A., & Earp, J. (2018) *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>
- Boers, E., Afzali, M. H., Newton, N., & Conrod, P. (2019). Association of screen time and depression in adolescence. *JAMA Pediatrics*, 173(9), 853–859.  
<https://doi.org/10.1001/jamapediatrics.2019.1759>
- Brackmann, C. P., Román-González, M., Robles, G., Moreno-León, J., Casali, A., & Barone, D. (2017). Development of computational thinking skills through unplugged activities in primary school. In E. Barendsen & P. Hubwieser (Eds.), *Proceedings of the 12th workshop in primary and secondary computing education (WiPSCE '17)* (pp. 65–72). ACM.  
<https://doi.org/10.1145/3137065.3137069>
- Brackmann, C. P., Barone, D. A. C., Boucinha, R. M., & Reichert, J. (2019). Development of computational thinking in Brazilian schools with social and economic vulnerability: How to teach computer science without machines. *International Journal of Innovation Education and Research*, 7(4), 79–96. <https://doi.org/10.31686/ijer.vol7.iss4.1390>
- 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.
- Brennan, K. A. (2013). *Best of both worlds: issues of structure and agency in computational creation, in an out of school* [Doctoral dissertation, Massachusetts Institute of Technology]. Massachusetts Institute of Technology. <https://dspace.mit.edu/handle/1721.1/79157>
- Brennan, K., Balch, C., & Chung, M. (2014) *Creative computing curriculum*. ScratchEd. <http://scratched.gse.harvard.edu/guide/curriculum.html>
- Buitrago Flórez, F., Casallas, R., Hernández, M., Reyes, A., Restrepo, S., & Danies, G. (2017). Changing a generation's way of thinking: Teaching computational thinking through programming. *Review of Educational Research*, 87(4), 834–860. <https://doi.org/10.3102%2F0034654317710096>
- Bull, G., Garofalo, J., & Hguyen, N. R. (2020). Thinking about computational thinking. *Journal of Digital Learning in Teacher Education*, 36(1), 6–18.  
<https://doi.org/10.1080/21532974.2019.1694381>
- 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.
- Campe, S., Denner, J., Green, D., & Torres, D. (2020). Pair programming in middle school: Variations in interactions and behaviors. *Computer Science Education*, 30(1), 22–46. <https://doi.org/10.1080/08993408.2019.1648119>
- 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.  
<https://doi.org/10.1016/j.ijcci.2019.06.005>

- Ch'ng, S. I., Low, Y. C., Lee, Y. L., Chia, W. C., & Yeong, L. S. (2019). Video games: A potential vehicle for teaching computational thinking. In S.-C.-Kong & H. Abelson (Eds.), *Computational thinking education* (pp. 247–260). Springer. [https://doi.org/10.1007/978-981-13-6528-7\\_14](https://doi.org/10.1007/978-981-13-6528-7_14)
- Chalmers, C. (2018). Robotics and computational thinking in primary school. *International Journal of Child-Computer Interaction*, 17, 93–100. <https://doi.org/10.1016/j.ijcci.2018.06.005>
- Çiftci, S., & Bildiren, A. (2020). The effect of coding courses on the cognitive abilities and problem-solving skills of preschool children. *Computer Science Education*, 30(1), 3–21. <https://doi.org/10.1080/08993408.2019.1696169>
- Città, G., Gentile, M., Allegra, M., Arrigo, M., Conti, D., Ottaviano, S., Reale, F., & Sciortino, M. (2019). The effects of mental rotation on computational thinking. *Computers & Education*, 141, Article No. 103613. <https://doi.org/10.1016/j.compedu.2019.103613>
- Coenraad, M., Hopcraft, C., Jozefowicz, J., Franklin, D., Palmer, J., & Weintrop, D. (2020). Helping teachers make equitable decisions: Effects of the TEC rubric on teachers' evaluations of a computing curriculum. *Computer Science Education*. <https://doi.org/10.1080/08993408.2020.1788862>
- Connor, R., Cutts, Q., & Robertson, J. (2017). Keeping the machinery in computing education. *Communications of the ACM*, 60(11), 26–28. <http://dx.doi.org/10.1145/3144174>
- Collins, A., Brown, J. S., & Newman, S. E. (2018). Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In L. B. Resnick (Ed.), *Knowing, learning, and instruction* (1st ed.), 42 pages. Routledge. <https://doi.org/10.4324/9781315044408-14>
- Cutumisu, M., Adams, C., & Lu, C. (2019). A scoping review of empirical research on recent computational thinking assessments. *Journal of Science Education and Technology*, 28, 651–676. <https://doi.org/10.1007/s10956-019-09799-3>
- Csizmadia, A., Curzon, P., Dorling, M., Humphreys, S., Ng, T., Selby, C., & Woollard, J. (2015). *Computational thinking. A guide for teachers*. Computing at school. <https://community.computingatschool.org.uk/files/6695/original.pdf>
- Dagienè, V., & Futschek, G. (2008). Bebras international contest on informatics and computer literacy: Criteria for good tasks. In R. T. Mittermeir & M. M. Syslo (Eds.), *ISSEP 2008. LNCS, vol. 5090* (pp. 19–30). Springer. [https://doi.org/10.1007/978-3-540-69924-8\\_2](https://doi.org/10.1007/978-3-540-69924-8_2)
- De Araujo, A. L. S. O., Andrade, W. L., & Guerrero, D. D. S. (2016). A systematic mapping study on assessing computational thinking abilities. In *Proceedings of the 2016 I.E. frontiers in education conference (FIE)* (pp. 1–9). IEEE. <https://doi.org/10.1109/FIE.2016.7757678>
- De Bruyckere, P., Kirschner, P. A., & Hulshof, C. (2020). *More urban myths about learning and education*. Routledge.



- De Paula, B. H., Valente, J. A., & Burn, A. (2014). Game-making as a means to deliver the new computing curriculum in England. *Currículo sem Fronteiras*, 14(3), 46–69.
- Deitrick, I., O'Connell, B., & Shapiro, R. B. (2014). The discourse of creative problem solving in childhood engineering education. In J. L. Polman, E. A. Kyza, D. K. O'Neill, I. Tabak, W. R. Penuel, A. S. Jurow, K. O'Connor, T. Lee, & L. D'Amico (Eds.), *Learning and becoming in practice: The international conference of the learning sciences (ICLS) 2014, Volume 2* (pp. 591–598). International Society of the Learning Sciences.  
<https://repository.isls.org/handle/1/1168>
- Del Olmo-Muñoz, J., Cózar-Gutiérrez, R., & González-Calero, J. A. (2020). Computational thinking through unplugged activities in early years of primary education. *Computers & Education*, 150, Article No. 103832.  
<https://doi.org/10.1016/j.compedu.2020.103832>
- Denner, J., Werner, L., Campe, S., & Ortiz, E. (2014). Pair programming: Under what conditions is it advantageous for middle school students? *Journal of Research on Technology in Education*, 46(3), 277–296.  
<https://doi.org/10.1080/15391523.2014.888272>
- Denner, J., Werner, L., & Ortiz, E. (2012). Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? *Computers & Education*, 58, 240–249.  
<https://doi.org/10.1016/j.compedu.2011.08.006>
- Denning, P., & Tedre, M. (2019). *Computational thinking*. MIT Press Ltd.
- Denning, P. (2017). Remaining trouble spots with computational thinking. *Communications of the ACM*, 60(6), 33–39. <https://doi.org/10.1145/2998438>
- Dohn, N. B. (2019). Students' interest in Scratch coding in lower secondary mathematics. *British Journal of Educational Technology*, 51(1), 71–83.  
<https://doi.org/10.1111/bjet.12759>
- Dong, Y., Cateté, V., Jocius, R., Lytle, N., Barnes, T., Albert, J., Joshi, D., Robinson, R., & Andrews, A. (2019). PRADA: A practical model for integrating computational thinking in K-12 education. In E. K. Hawthorne & M. A. Pérez-Quiñones (Eds.), *Proceedings of the 50th ACM technical symposium on computer science education (SIGCSE '19)* (pp. 906–912). ACM.  
<https://doi.org/10.1145/3287324.3287431>
- Duit, R., & Treagust, D. F. (2003). Conceptual change: A powerful framework for improving science teaching and learning. *International Journal of Science Education*, 25(6), 671–688. <https://doi.org/10.1080/09500690305016>
- Dufva, T., & Dufva, M. (2019). Grasping the future of the digital society. *Futures*, 107, 17–28. <https://www.doi.org/10.1016/j.futures.2018.11.001>
- Duncan, C., & Bell, T. (2015). A pilot computer science and programming course for primary school students. In J. Gal-Ezer, S. Sentence & J. Vahrenhold (Eds.), *Proceedings of the workshop in primary and secondary computing education (WiPSCE '15)* (pp. 39–48). ACM.  
<https://doi.org/10.1145/2818314.2818328>

- Durak, H. Y., & Saritepeci, M. (2018). Analysis of the relation between computational thinking skills and various variables with the structural equation model. *Computers & Education*, *116*, 191–202.  
<https://doi.org/10.1016/j.compedu.2017.09.004>
- Dwyer, H., Hill, C., Carpenter, S., Harlow, D., & Franklin, D. (2014). Identifying elementary students' pre-instructional ability to develop algorithms and step-by-step instructions. In J. D. Dougherty & K. Nagel (Eds.), *Proceedings of the 45th ACM technical symposium on computer science education (SIGCSE '14)* (pp. 511–516). ACM.  
<https://doi.org/10.1145/2538862.2538905>
- Falkner, K., Sentance, S., Vivian, R., Barksdale, S., Busuttil, L., Cole, E., Liebe, C., Maiorana, F., McGill, M. M., & Quille, K. (2019). An international comparison of K–12 computer science education intended and enacted curricula. In P. Ihantola & N. Falkner (Eds.), *Proceedings of the 19th Koli calling international conference on computing education research (Koli calling '19)* (Article No. 4). ACM. <http://doi.org/10.1145/3364510.3364517>
- Franklin, D., Coenraad, M., Palmer, J., Eatinger, D., Zipp, A., Anaya, M., White, M., Pham, H., Gökdemir, O., & Weintrop, D. (2020b). An analysis of use-modify-create pedagogical approach's success in balancing structure and student agency. In A. Robins, A. Moskal, A. J. Ko. & R. McCauley (Eds.), *Proceedings of the 2020 ACM conference on international computing education research (ICER '20)* (pp. 14–24). ACM.  
<https://doi.org/10.1145/3372782.3406256>
- Franklin, D., Conrad, P., Boe, B., Nilsen, K., Hill, C., Len, M., Dreschler, G., Aldana, G., Almeida-Tanaka, P., Kiefer, B., Laird, C., Lopez, F., Pham, C., Suarez, J., & Waite, R. (2013). Assessment of computer science learning in a Scratch-based outreach program. In T. Camp & P. Tymann (Eds.), *Proceedings of the 44th ACM technical symposium on computer science education (SIGCSE '13)* (pp. 371–376). ACM.  
<https://doi.org/10.1145/2445196.2445304>
- Franklin, D., Hill, C., Dwyer, H. A., Hansen, A. K., Iveland, A., & Harlow, D. B. (2016). Initialization in Scratch: Seeking knowledge transfer. In A. Alphonse & J. Tims (Eds.), *Proceedings of the 47th ACM technical symposium on computing science education (SIGCSE '16)* (pp. 217–222). ACM.  
<https://doi.org/10.1145/2839509.2844569>
- Franklin, D., Salac, J., Crenshaw, Z., Turimella, S., Klain, Z., Anaya, M., & Thomas, C. (2020a). Exploring student behavior using the TIPP&SEE learning strategy. In A. Robins, A. Moskal, A. J. Ko. & R. McCauley (Eds.), *Proceedings of the 2020 ACM conference on international computing education research (ICER '20)* (pp. 91–101). ACM.  
<https://doi.org/10.1145/3372782.3406257>
- Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., Weintrop, D., & Harlow, D. (2017). Using upper-elementary student performance to understand conceptual sequencing in a blocks-based curriculum. In M. E. Caspersen & S. H. Edwards (Eds.), *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*

- (SIGCSE '17) (pp. 231–236). ACM.  
<https://doi.org/10.1145/3017680.3017760>
- Frailon, J., Ainley, J., Schulz, W., Friedman, T., & Duckworth, D. (2020). *Preparing for life in a digital world. IEA international computer and information literacy study 2018 international report*. Springer.  
<https://doi.org/10.1007/978-3-030-38781-5>
- Funke, A., & Geldreich, K. (2017). Measurement and visualization of programming processes of primary school students in Scratch. In E. Barendsen & P. Hubwieser (Eds.), *Proceedings of the 12th workshop in primary and secondary computing education (WiPSCE '17)* (pp. 101–102). ACM. <https://doi.org/10.1145/3137065.3137086>
- 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 (EDUCON), Athens, Greece. <https://doi.org/10.1109/EDUCON.2017.7943005>
- Gane, B. D., Israel, M., Elagha, N., Yan, W., Luo, F., & Pellegrino, J. W. (2021). Design and validation of learning-trajectory based assessments for computational thinking in upper elementary grades. *Computer Science Education*, 31(2), 141–168. <https://doi.org/10.1080/08993408.2021.1874221>
- 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 (EDUCON), Tallinn, Estonia. <http://doi.org/10.1109/EDUCON.2015.7096023>
- Georgina, M. T., Ojo, S. O., & Lall, M. (2015). a conceptual metaphor based model for enhanced understanding of programming concepts. In *2015 proceedings of the EDSIG conference* (Article No. 3480). ISCAP.
- Gibson, B., & Bell, T. (2013). Evaluation of games for teaching computer science. In M. E. Caspersen, M. Knobelsdorf & R. Romeike (Eds.), *Proceedings of the 8th workshop in primary and secondary computing education (WiPSCE '13)* (pp. 51–60). ACM. <https://doi.org/10.1145/2532748.2532751>
- Giordano, D., Maiorana, F., Csizmadia, A., Marsden, S. (2015). New horizons in the assessment of computer science at school and beyond: Leveraging on the ViVA platform. In N. Ragonis, & P. Kinnunen (Ed.s) *Proceedings of the 2015 ITiCSE on working group reports (ITiCSE'15)* (pp. 117–147). ACM. <https://doi.org/10.1145/2858796.2858801>
- Given, L. M. (2008). *The SAGE encyclopedia of qualitative research methods (vols. 1-0)*. SAGE. <http://dx.doi.org/10.4135/9781412963909.n194>
- Grover, S. (2018). A Tale of Two CTs (and a Revised Timeline for Computational Thinking). *Communications of the ACM blog (Blog@CACM)*. <https://cacm.acm.org/blogs/blog-cacm/232488-a-tale-of-two-cts-and-a-revised-timeline-for-computational-thinking/fulltext>
- Grover, S. (2020). Designing an assessment for introductory programming concepts in middle school computer science. In J. Zhang, & M. Sherriff (Eds). *Proceedings of the 51st ACM technical symposium on computer science*

- education (SIGCSE '20)* (pp. 678–684). ACM.  
<https://doi.org/10.1145/3328778.3366896>
- Grover, S., & Basu, S. (2017). Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In M. E. Caspersen & S. H. Edwards (Eds.), *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education (SIGCSE '17)* (pp. 267–272). ACM.  
<https://doi.org/10.1145/3017680.3017723>
- 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 A. Wise, P. H. Winne & G. Lynch (Eds.), *Proceedings of the seventh international learning analytics & knowledge conference (LAK '17)* (pp. 530–531). ACM. <https://doi.org/10.1145/3105910>
- Grover, S., Jackiw, N., & Lundh, P. (2019). Concepts before coding: non-programming interactives to advance learning of introductory programming concepts in middle school. *Computer Science Education*, 29(2–3), 106–135. <http://doi.org/10.1080/08993408.2019.1568955>
- Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher*, 41(1), 38–43.  
<http://doi.org/10.3102/0013189X12463051>
- Grover, S., & Pea, R. (2018). Computational thinking: A competency whose time has come. In S. Sentence, E. Barendsen & C. Schulte (Eds.), *Computer science education: Perspectives on teaching and learning in school* (pp. 19–37). Bloomsbury Academic.
- Hameed, S., Low, C.-W., Lee, P.-T., Mohamed, N., Ng, W.-B., Seow, P., & Wadhwa, B. (2018). A school-wide approach to infusing coding in the curriculum. In S. C. Kong, D. Andone, G. Biswas, T. Crick, H. U. Hoppe, T. C. Hsu, R. H. Huang, K. Y. Li, C. K. Looi, M. Milrad, J. Sheldon, J. L. Shih, K. F. Sin, M. Tissenbaum & J. Vahrenhold (Eds.), *Proceedings of the 2nd computational thinking education conference 2018 (CTE2018)* (pp. 33–36). The Education University of Hong Kong.
- Hao, Q., Smith IV, D. H., Ding, L., Ko, A., Ottaway, C., Wilson, J., Arakawa, K. H., Turcan, A., Poehlman, T., & Greer, T. (2021). Towards understanding the effective design of automated formative feedback for programming assignments. *Computer Science Education*.  
<https://doi.org/10.1080/08993408.2020.1860408>
- Haroldson, R., & Ballard, D. (2020). Alignment and representation in computer science: an analysis of picture books and graphic novels for K–8 students. *Computer Science Education*, 31(1), 4–29.  
<https://doi.org/10.1080/08993408.2020.1779520>
- Heintz, F., Mannila, L., & Färnqvist, T. (2016). A review of models for introducing computational thinking, computer science and computing in K–12 education. In *2016 IEEE frontiers in education conference (FIE)* (pp. 1–9). IEEE. <https://doi.org/10.1109/FIE.2016.7757410>.

- Hoppe, H. U., & Werneburg, S. (2019). Computational thinking – More than a variant of scientific inquiry! In S. C. Kong & H. Abelson (Eds.), *Computational thinking education* (pp. 13–30). Springer.  
[https://doi.org/10.1007/978-981-13-6528-7\\_2](https://doi.org/10.1007/978-981-13-6528-7_2)
- 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 & Education*, 126, 296–310.  
<https://doi.org/10.1016/j.compedu.2018.07.004>
- Hu, C. (2011). Computational Thinking – What it might mean and what we might do about it. In G. Rößling (Ed.), *Proceedings of the 16th annual joint conference on innovation and technology in computer science education (ITiCSE'11)* (pp. 223–227). ACM. <http://doi.org/10.1145/1999747.1999811>
- Huang, W., & Looi, C.-K. (2020). A critical review of literature on “unplugged” pedagogies in K-12 computer science and computational thinking education. *Computer Science Education*, 31(1), 83–111.  
<https://doi.org/10.1080/08993408.2020.1789411>
- Hubbard, A. (2018). Pedagogical content knowledge in computing education: a review of the research literature. *Computer Science Education*, 28(2), 117–135. <https://doi.org/10.1080/08993408.2018.1509580>
- Hutchins, N., Biswas, G., Conlin, L. D., & Emara, M. (2018). Studying synergistic learning of physics and computational thinking in a learning by modeling environment. In J. C. Yang, M. Chang, L.-H. Wong & M. M. T. Rodrigo (Eds.), *Proceedings of the 26th international conference on computers in education (ICCE 2018)* (pp. 153–162). Asia-Pacific Society for Computers in Education.
- Höfer, A. (2008). Video analysis of pair programming. In P. Kruchten & S. Adolph (Eds.), *Proceedings of the 2008 international workshop on scrutinizing agile practices or shoot-out at the agile corral (APOS '08)* (pp. 37–41). ACM.  
<https://doi.org/10.1145/1370143.1370151>
- Høholt, M., Graungaard, D., Bouvin, N. O., Petersen, M. G., & Eriksson, E. (2021). Towards a model of progression in computational empowerment in education. *International Journal of Child-Computer Interaction*, 29, Article No. 100302. <https://doi.org/10.1016/j.ijcci.2021.100302>
- Isbell, C. L., Stein, L. A., Cutler, R., Forbes, J., Fraser, L., Impagliazzo, J., Proulx, V., Russ, S., Thomas, R., Xu, Y. (2009). (Re)defining computing curricula by (re)defining computing. In P. Brézillon (Ed.), *Proceedings of the 14th annual ACM SIGCSE conference on innovation and technology in computer science education (ITiCSE '09)* (pp. 195–207). ACM.  
<https://doi.org/10.1145/2481449.2481466>
- Israel, M., Wherfel, Q. M., Shehab, S., Ramos, E. A., Metzger, A., & Reese, G. C. (2016). Assessing collaborative computing: Development of the collaborative-computing observation instrument (C-COI). *Computer Science Education*, 26(2–3), 208–233.  
<https://doi.org/10.1080/08993408.2016.1231784>

- Israel-Fishelson, R., HersHKovitz, A., Eguíluz, A., Garaizar, P., & Guenaga, M. (2020). Computational thinking and creativity: A test for interdependency. In S.-C. Kong, H. U. Hoppe, T.-C. Hsu, R.-H. Huang, B.-C. Kuo, R. K.-Y. Li, C.-K. Looi, M. Milrad, J.-L. Shih, K.-F. Sin, K.-S. Song, M. Specht, F. Sullivan & J. Vahrenhold (Eds.), *Proceedings of the international conference on computational thinking education 2020 (CTE2020)* (pp. 15–20). The Education University of Hong Kong.
- ISTE 2018 = International Society for Technology in Education (2018). *Computational thinking competencies*. International Society for Technology in Education. <https://www.iste.org/standards/computational-thinking>
- Iwata, M., Pitkänen, K., Laru, J., & Mäkitalo, K. (2020). Exploring potentials and challenges to develop twenty-first century skills and computational thinking in K–12 maker education. *Frontiers in Education, 5*, 1–16. <https://doi.org/10.3389/feduc.2020.00087>
- Jacob, S. R., & Warschauer, M. (2018). Computational thinking and literacy. *Journal of Computer Science Integration, 1*(1), Article No. 1. <http://doi.org/10.26716/jcsi.2018.01.1.1>
- Jun, S., Han, S., Kim, H., & Lee, W. (2014). Assessing the computational literacy of elementary students on a national level in Korea. *Educational Assessment, Evaluation and Accountability, 26*, 319–332. <https://doi.org/10.1007/s11092-013-9185-7>
- Kafai, Y. B., & Burke, Q. (2013a). Computer programming goes back to school. *Phi Delta Kappan, 95*(1), 61–65. <https://doi.org/10.1177%2F003172171309500111>
- Kafai, Y. B., & Burke, Q. (2013b). The social turn in K–12 programming: Moving from computational thinking to computational participation. In T. Camp & P. Tymann (Eds.), *Proceedings of the 44th ACM technical symposium on computer science education (SIGCSE '13)* (pp. 603–608). ACM. <https://doi.org/10.1145/2445196.2445373>
- Kafai, Y., Proctor, C., & Lui, D. (2019). From theory bias to theory dialogue: Embracing cognitive, situated, and critical framings of computational thinking in K–12 CS education. In L. Malmi, A. Korhonen, R. McCartney & A. Petersen (Eds.), *Proceedings of the 2019 ACM conference on international computing education research (ICER '18)* (pp. 101–109). ACM. <https://doi.org/10.1145/3291279.3339400>
- Kahn, K., Sendova, E., Sacristán, A. I., & Noss, R. (2011). Young students exploring cardinality by constructing infinite processes. *Technology, Knowledge, and Learning, 16*(1), 3–34. <https://doi.org/10.1007/s10758-011-9175-0>
- Kalelioğlu, F., Gülbahar, Y., & Kukul, V. (2016). A framework for computational thinking based on a systematic research review. *Baltic Journal of Modern Computing, 4*(3), 583–596.
- Kazimoglu, C., Kiernan, M., Bacon, L., & MacKinnon, L. (2012). Learning programming at the computational thinking level via digital game-play.

- Procedia Computer Science*, 9, 522–531.  
<https://doi.org/10.1016/j.procs.2012.04.056>
- Ke, F. (2013). An implementation of design-based learning through creating educational computer games: A case study on mathematics learning during design and computing. *Computers & Education*, 73, 26–39.  
<https://doi.org/10.1016/j.compedu.2013.12.010>
- Keeney, P. (2018). *Science formative assessment. 75 practical strategies for linking assessment, instruction, and learning*. SAGE.
- Kirschner, P. A., Sweller, J., & Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, 41(2), 75–86.  
[https://doi.org/10.1207/s15326985ep4102\\_1](https://doi.org/10.1207/s15326985ep4102_1)
- Klahr, D., & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology*, 20, 362–404. [https://psycnet.apa.org/doi/10.1016/0010-0285\(88\)90004-7](https://psycnet.apa.org/doi/10.1016/0010-0285(88)90004-7)
- Koh, K. H., Basawapatna, A., Bennett, V., & Repenning, A. (2010). Towards the automatic recognition of computational thinking for adaptive visual language learning. In *Proceedings of the 2010 IEEE symposium on visual languages and human-centric computing (VL/HCC)* (pp. 59–66). IEEE.  
<https://doi.org/10.1109/VLHCC.2010.17>
- Koh, K. H., Basawapatna, A., Nickerson, H., & Repenning, A. (2014). Real time assessment of computational thinking. In *Proceedings of the 2014 IEEE symposium on visual languages and human-centric computing (VL/HCC)* (pp. 49–52). IEEE. <http://doi.org/10.1109/VLHCC.2014.6883021>
- Kong, S.-C. (2016). A framework of curriculum design for computational thinking development in K-12 education. *Journal of Computers in Education*, 3(4), 377–394. <https://doi.org/10.1007/s40692-016-0076-z>
- Kong, S.-C. (2019). Components and methods of evaluating computational thinking for fostering creative problem-solvers in senior primary school education. In S.-C. Kong & H. Abelson (Eds.), *Computational thinking education* (pp. 119–144). Springer. [https://doi.org/10.1007/978-981-13-6528-7\\_8](https://doi.org/10.1007/978-981-13-6528-7_8)
- Kong, S.-C., Chiu, M. M., Lai, M. (2018). A study of primary school students' interest, collaboration attitude, and programming empowerment in computational thinking education. *Computers & Education*, 127, 178–189.  
<https://doi.org/10.1016/j.compedu.2018.08.026>
- Kong, S.-C., Liu, M., & Sun, D. (2020). Teacher development in computational thinking: design and learning outcomes of programming concepts, practices and pedagogy. *Computers & Education*, 151, Article No. 103872.  
<https://doi.org/10.1016/j.compedu.2020.103872>
- Korhonen, T., & Lavonen, J. (2016). A new wave of learning in Finland: Get started with innovation! In S. Choo, D. Sawch, A. Villanueva & R. Vinz (Eds.), *Educating for the 21st century* (pp. 447–467). Springer.  
[https://doi.org/10.1007/978-981-10-1673-8\\_24](https://doi.org/10.1007/978-981-10-1673-8_24)

- Korhonen, T. (2017). *Kodin ja koulun digitaalinen kumppanuus* [Doctoral dissertation, University of Helsinki]. University of Helsinki. <http://urn.fi/URN:ISBN:978-951-51-3256-7>
- Labusch, A., Eickelmann, B., & Vennemann, M. (2019). Computational thinking processes and their congruence with problem-solving and information processing. In S.-C. Kong & H. Abelson (Eds.), *Computational thinking education* (pp. 65–78). Springer. [https://doi.org/10.1007/978-981-13-6528-7\\_5](https://doi.org/10.1007/978-981-13-6528-7_5)
- Lamprou, A., & Repenning, A. (2018). Teaching how to teach computational thinking. In I. Polycarpou & J. C. Read (Eds.), *Proceedings of the 23rd annual ACM conference on innovation and technology in computer science education (ITiCSE '18)* (pp. 69–74). ACM. <https://doi.org/10.1145/3197091.3197120>
- 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. <https://doi.org/10.1145/1929887.1929902>
- Leino, K., Rikala, J., Puhakka, E., Niilo-Rämä, M., Sirén, M., & Fagerlund, J. (2019). *Digiloikasta digitaitoihin. Kansainvälinen monilukutaidon ja ohjelmoinnillisen ajattelun tutkimus (ICILS 2018)*. Koulutuksen tutkimuslaitos: Jyväskylän yliopisto. <https://jyx.jyu.fi/handle/123456789/66250>
- Lewis, C. M. (2011). Is pair programming more effective than other forms of collaboration for young students? *Computer Science Education*, 21(2), 105–134. <https://doi.org/10.1080/08993408.2011.579805>
- Lewis, C., & Shah, N. (2015). How equity and inequity can emerge in pair programming. In B. Dorn (Ed.), *Proceedings of the eleventh annual international conference on international computing education research (ICER '15)* (pp. 41–50). ACM. <https://doi.org/10.1145/2787622.2787716>
- Liebe, C., & Camp, T. (2019). An examination of abstraction in K–12 computer science education. In P. Ihantola & N. Falkner (Eds.), *Proceedings of the 19th Koli calling international conference on computing education research (Koli calling '19)* (Article No. 9). ACM. <http://doi.org/10.1145/3364510.3364526>
- Lin, Q., Yin, Y., Tang, X., Hadad, R., & Zhai, X. (2020). Assessing learning in technology-rich maker activities: A systematic review of empirical research. *Computers & Education*, 157, 103944. <https://doi.org/10.1016/j.compedu.2020.103944>
- Liu, X., Zhi, R., Hicks, A., & Barnes, T. (2017). Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education*, 27(1), 1–29. <http://doi.org/10.1080/08993408.2017.1308651>
- Liukas, L., & Mykkänen, J. (2014). *Koodi2016: Ensiapua ohjelmoinnin opetukseen peruskoulussa*. [https://s3-eu-west-1.amazonaws.com/koodi2016/Koodi2016\\_LR.pdf](https://s3-eu-west-1.amazonaws.com/koodi2016/Koodi2016_LR.pdf)
- Lonka, K., Kruskopf, M., & Hietajärvi, L. (2018). Competence 5: Information and communication technology (ICT). In K. Lonka (Ed.), *Phenomenal Learning from Finland* (pp. 129–150). Edita.



- Looi, C.-K., How, M.-L., Longkai, W., Seow, P., & Liu, L. (2018). Analysis of linkages between an unplugged activity and the development of computational thinking. *Computer Science Education, 28*(3), 255–279. <https://doi.org/10.1080/08993408.2018.1533297>
- Lu, J. J., & Fletcher, G. H. L. (2009). Thinking about computational thinking. *SIGCSE Bulletin, 41*(1), 260–264. <https://doi.org/10.1145/1539024.1508959>
- Lui, D., Jayathirtha, G., Fields, D. A., Shaw, M., & Kafai, Y. B. (2018). Design considerations for capturing computational thinking practices in high school students' electronic textile portfolios. In Y. B. Kafai, W. A. Sandoval & N. Enyedy (Eds.), *Proceedings of the international conference of the learning sciences (ICLS '18)* (pp. 721–728). University College London.
- Luo, F., Antonenko, P. D., & Davis, E. C. (2020). Exploring the evolution of two girls' conceptions and practices in computational thinking in science. *Computers & Education, 146*, Article No. 103759. <https://doi.org/10.1016/j.compedu.2019.103759>
- Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K–12? *Computers in Human Behavior, 41*, 51–61. <https://doi.org/10.1016/j.chb.2014.09.012>
- Mack, L. (2010). The philosophical underpinnings of educational research. *Polyglossia, 19*, 5–11.
- Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. *SIGCSE Bulletin, 39*(1), 223–227. <https://doi.org/10.1145/1227310.1227388>
- Maloney, J., Peppler, K., Kafai, Y. B., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with Scratch. In J. D. Dougherty & S. Rodger (Eds.), *Proceedings of the 39th SIGCSE technical symposium on computer science education (SIGCSE '08)* (pp. 367–371). ACM. <https://doi.org/10.1145/1352135.1352260>
- 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. <https://doi.org/10.1145/1868358.1868363>
- Manches, A., McKenna, P. E., Rajendran, G., & Robertson, J. (2020). Identifying embodied metaphors for computing education. *Computers in Human Behavior, 105*, Article No. 105859. <https://doi.org/10.1016/j.chb.2018.12.037>
- Mannila, L., Dagiene, V., Demo, B., Grgurina, N., Mirolo, C., Rolandsson, L., & Settle, A. (2014). Computational thinking in K–9 education. In A. Clear & R. Lister (Eds.), *Proceedings of the working group reports of the 2014 on innovation & technology in computer science education conference (ITiCSE '14)* (pp. 1–29). ACM. <https://doi.org/10.1145/2713609.2713610>
- Mannila, L., Heintz, F., Kjällander, S., & Åkerfeldt, A. (2020). Programming in primary education: Towards a research based assessment framework. In T. Brinda & M. Armoni (Eds.), *Proceedings of the 15th workshop in primary and*

- secondary computing education (WiPSCE '20)* (Article No. 10). ACM.  
<https://doi.org/10.1145/3421590.3421598>
- Mannila, L., Nordén, L.-Å., & Pears, A. (2018). Digital competence, teacher self-efficacy and training needs. In L. Malmi, A. Korhonen, R. McCartney & A. Petersen (Eds.), *Proceedings of the 2018 ACM conference on international computing education research (ICER '18)* (pp. 78–85). ACM.  
<https://doi.org/10.1145/3230977.3230993>
- Martin, F. (2018). *Rethinking computational thinking*. Computer Science Teachers Association. <http://advocate.csteachers.org/2018/02/17/rethinking-computational-thinking/>
- Mason, S. L., & Rich, P. J. (2020). Development and analysis of the elementary student coding attitudes survey. *Computers & Education*, 153, Article No. 103898. <https://doi.org/10.1016/j.compedu.2020.103898>
- Mayer, R. E. (2004). Should there be a three-strikes rule against pure discovery learning? The case for guided methods of instruction. *American Psychologist*, 59(1), 14–19. <https://doi.org/10.1037/0003-066x.59.1.14>
- Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). Learning computer science concepts with Scratch. *Computer Science Education*, 23(3), 239–264. <https://doi.org/10.1145/1839594.1839607>
- Mertala, P. (2018). Young children's conceptions of computers, code, and the Internet. *International Journal of Child-Computer Interaction*, 19, 56–66. <http://doi.org/10.1016/j.ijcci.2018.11.003>
- Mertala, P., Palsa, L., & Dufva, T. S. (2020). Monilukutaito koodin purkajana: Ehdotus laaja-alaiseksi ohjelmoinnin pedagogiikaksi. *Media & Viestintä*, 43(1), 21–46. <https://doi.org/10.23983/mv.91079>
- Michaelson, G. (2015). Teaching programming with computational and informational thinking. *Journal of Pedagogic Development*, 5(1), 51–66.
- Mladenović, M., Boljat, I., & Žanko, Ž. (2018). Comparing loops misconceptions in a block-based and text-based programming languages at the K–12 level. *Education and Information Technologies*, 23(4), 1483–1500. <https://doi.org/10.1007/s10639-017-9673-3>
- Moreno-León, J., & Robles, G. (2016). *Code to learn with Scratch? A systematic literature review*. Paper presented at the 2016 IEEE global engineering education conference (EDUCON), Abu Dhabi, United Arab Emirates. <https://doi.org/10.1109/EDUCON.2016.7474546>
- 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. <https://doi.org/10.6018/red/46/10>
- 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*. <http://doi.org/10.1109/TETC.2017.2734818>
- Moschella, M. (2019). Observable computational thinking skills in primary school children: How and when teachers can discern abstraction,

- decomposition and use of algorithms. In I. Gómez Chova, A. López Martínez & I. Candel Torres (Eds.), *Proceedings of the 13th international technology, education and development conference (INTED 2019)* (pp. 6259–6267). IATED. <http://doi.org/10.21125/inted.2019.1523>
- Mäkitalo, K., Kohnle, C., & Fischer, F. (2011). Computer-supported collaborative inquiry learning and classroom scripts: Effects on help-seeking processes and learning outcomes. *Learning & Instruction, 21*(2), 257–266. <https://doi.org/10.1016/j.learninstruc.2010.07.001>
- Mäkitalo, K. H., Tedre, M., Laru, J., & Valtonen, T. (2019). Computational thinking in Finnish pre-service teacher education. In S. C. Kong, D. And one, G. Biswas, G. Biswas, H. U. Hoppe, T. C. Hsu, R. H. Huang, B. C. Kuo, K. Y. Li, C. K. Looi, M. Milrad, J. Sheldon, J. L. Shih, K. F. Sin, K. S. Song & J. Vahrenhold (Eds.), *Proceedings of the international conference on computational thinking education 2019 (CTE2019)* (pp. 105–108). The Education University of Hong Kong.
- Navarro-Prieto, R., & Cañas, J. J. (2001). Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human-Computer Studies, 54*, 799–829. <https://doi.org/10.1006/ijhc.2000.0465>
- Niemelä, P. (2018). *From Legos and Logos to Lambda: A Hypothetical Learning Trajectory for Computational Thinking* [Doctoral dissertation, Tampere University of Technology]. Tampere University of Technology. [https://tutcris.tut.fi/portal/files/16515265/niemela\\_1565.pdf](https://tutcris.tut.fi/portal/files/16515265/niemela_1565.pdf)
- Nijenhuis-Voogt, J., Bayram-Jacobs, D., Meijer, P. C., & Barendsen, E. (2020). Omnipresent yet elusive: Teachers' views on contexts for teaching algorithms in secondary education. *Computer Science Education, 31*(1), 30–59. <https://doi.org/10.1080/08993408.2020.1783149>
- Oleson, A., Wortzman, B., & Ko, A. J. (2020). On the role of design in K–12 computing education. *ACM Transactions on Computing Education, 21*(1), Article No. 2. ACM. <https://doi.org/10.1145/3427594>
- Opetushallitus (2014). *Perusopetuksen opetussuunnitelman perusteet 2016*. Opetushallitus.
- Ota, G., Morimoto, Y., & Kato, H. (2016). *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. <https://doi.org/10.1109/VLHCC.2016.7739695>
- Papadakis, S. J., Kalogiannakis, M., Orfanakis, V., & Zaranis, N. (2017). The appropriateness of Scratch and App Inventor as educational environments for teaching introductory programming in primary and secondary education. *International Journal of Web-Based Learning and Teaching Technologies, 12*(4), Article No. 6. <https://doi.org/10.4018/IJWLTT.2017100106>
- Papadakis, S. J., Kalogiannakis, M., & Zaranis, N. (2016). Developing fundamental programming concepts and computational thinking with

- ScratchJr in preschool education: A case study. *International Journal of Mobile Learning and Organisation*, 10(3), 187–202.  
<https://doi.org/10.1504/IJMLO.2016.077867>
- Papert, S., & Harel, I. (1991). Situating constructionism. In S. Papert & I. Harel (Eds.), *Constructionism* (pp. 1–11). Ablex Publishing.
- Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.
- Park, Y., & Shin, Y. (2019). Comparing the effectiveness of Scratch and App Inventor with regard to learning computational thinking concepts. *Electronics*, 8(11), Article No. 1269.  
<https://doi.org/10.3390/electronics8111269>
- Parsons, D., & Haden, P. (2007). Programming osmosis: Knowledge transfer from imperative to visual programming environments. In S. Mann & N. Bridgeman (Eds.), *Proceedings of the 20th annual conference of the national advisory committee on computing qualifications (NACCQ '07)* (pp. 209–215). NACCQ.
- Patton, E. W., Tissenbaum, M., & Harumani, F. (2019). MIT App Inventor: Objectives, design, and development. In S.-C. Kong & H. Abelson (Eds.), *Computational thinking education* (pp. 31–49). Springer.  
[https://doi.org/10.1007/978-981-13-6528-7\\_3](https://doi.org/10.1007/978-981-13-6528-7_3)
- Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology*, 2(2), 137–168.  
[http://doi.org/10.1016/0732-118X\(84\)90018-7](http://doi.org/10.1016/0732-118X(84)90018-7)
- Pea, R. D., Soloway, E., & Sphorer, J. C. (1987). The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics*, 9(1), 5–30.
- Pears, A., Barendsen, E., & Dagienè, V. (2019). Holistic STEAM education through computational thinking: A perspective on training future teachers. In S. N. Pozdniakov & V. Dagienè (Eds.), *Informatics in schools. new ideas in school informatics. lecture notes in computer science, vol. 11913*. Springer. [https://doi.org/10.1007/978-3-030-33759-9\\_4](https://doi.org/10.1007/978-3-030-33759-9_4)
- Pelánek, R., & Effenberger, T. (2020). Design and analysis of microworlds and puzzles for block-based programming. *Computer Science Education*. doi: <https://doi.org/10.1080/08993408.2020.1832813>
- Pérez-Marín, D., Hijón-Neira, R., Bacelo, A., & Pizarro, C. (2020). Can computational thinking be improved by using a methodology based on metaphors and Scratch to teach computer programming to children? *Computers in Human Behavior*, 105, Article No. 105849.  
<https://doi.org/10.1016/j.chb.2018.12.027>
- Perković, L., Settle, A., Hwang, S., & Jones, J. (2010). A framework for computational thinking across the curriculum. In R. Ayfer & J. Impagliazzo (Eds.), *Proceedings of the 15th annual conference on innovation and technology in computer science education (ITiCSE '10)* (pp. 123–127). ACM. <http://doi.org/10.1145/1822090.1822126>

- Popat, S., & Starkey, L. (2018). Learning to code or coding to learn? A systematic review. *Computers & Education*, 128, 365–376.  
<https://doi.org/10.1016/j.compedu.2018.10.005>
- Preston, D. (2005). Pair programming as a model of collaborative learning: A review of the research. *Journal of Computing Sciences in Colleges*, 4, 39–45.
- Promraksa, S., Sangaroon, K., & Inprasitha, M. (2014). Characteristics of computational thinking about the estimation of the students in mathematics classroom applying lesson study and open approach. *Journal of Education and Learning*, 3(3), 56–66. <http://doi.org/10.5539/jel.v3n3p56>
- Redecker, C., & Punie, Y. (2017). *European framework for the digital competence of educators*. DigcompEdu. EUR – Scientific and Technical Research Reports. Publications Office of the European Union.  
<https://doi.org/10.2760/159770>
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60–67. <https://doi.org/10.1145/1592761.1592779>
- Resnick, M., Kafai, Y., & Maeda, J. (2003). *A networked, media-rich programming environment to enhance technological fluency at after-school centers in economically disadvantaged communities*. Proposal submitted to National Science Foundation.
- Resnick, M., Ocko, S., & Papert, S. (1988). Lego, Logo, and design. *Children's Environments Quarterly*, 5(4), 14–18.
- Rich, K. M., Binkowski, T. A., Strickland, C., & Franklin D. (2018). Decomposition: A K–8 computational thinking learning trajectory. In M. Joy & P. Ihantola (Eds.), *Proceedings of the 18th Koli calling international conference on computing education research (Koli calling '18)* (pp. 124–132). ACM. <https://doi.org/10.1145/3230977.3230979>
- Rich, K. M., Franklin, D., Strickland, C., Isaacs, A., & Etinger, D. (2020). A learning trajectory for variables based in computational thinking literature: Using levels of thinking to develop instruction. *Computer Science Education*. <https://doi.org/10.1080/08993408.2020.1866938>
- Rich, P. J., Mason, S. L., & O'Leary, J. (2021). Measuring the effect of continuous professional development on elementary teachers' self-efficacy to teach coding and computational thinking. *Computers & Education*, Article No. 104196. <https://doi.org/10.1016/j.compedu.2021.104196>
- Robles, G., Hauck, J. C. R., Román-González, M., & von Wangenheim, C. G. (2018). On tools that support the development of computational thinking skills: Some thoughts and future vision. In S. C. Kong, D. Andone, G. Biswas, T. Crick, H. U. Hoppe, T. C. Hsu, R. H. Huang, K. Y. Li, C. K. Looi, M. Milrad, J. Sheldon, J. L. Shih, K. F. Sin, M. Tissenbaum & J. Vahrenhold (Eds.), *Proceedings of the international conference on computational thinking education 2018 (CTE2018)* (pp. 129–132). The Education University of Hong Kong.

- Román-González, M., Moreno-León, J., & Robles, G. (2019). Complementary tools for computational thinking assessment. In S.-C. Kong & H. Abelson (Eds.), *Computational thinking education* (pp. 79-98). Springer.
- Román-González, M., Pérez-González, J.-C., & Jiménez-Fernández, C. (2017a). Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. *Computers in Human Behavior*, *72*, 678-691. <https://doi.org/10.1016/j.chb.2016.08.047>
- Román-González, M., Pérez-González, J.-C., Moreno-León, J., & Robles, G. (2017b). Extending the nomological network of computational thinking with non-cognitive factors. *Computers in Human Behavior*, *80*, 441-459. <https://doi.org/10.1016/j.chb.2017.09.030>
- Roschelle, J., & Teasley, S. D. (1995). The construction of shared knowledge in collaborative problem solving. In C. E. O'Malley (Ed.), *Computer-supported collaborative learning* (pp. 69-197). Springer. [https://doi.org/10.1007/978-3-642-85098-1\\_5](https://doi.org/10.1007/978-3-642-85098-1_5)
- 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. <https://doi.org/10.1016/j.compedu.2016.03.003>
- Salomon, G., & Perkins, D. N. (1985). Transfer of cognitive skills from programming: When and how? *Journal of Educational Computing Research*, *3*, 149-169. <https://doi.org/10.2190%2F6F4Q-7861-QWA5-8PL1>
- Saqr, M., Ng, K., Oyelere, S. S., & Tedre, M. (2021). People, ideas, milestones: A Scientometric study of computational thinking. *ACM Transactions on Computing Education*, *21*(3), Article No. 20 (pp. 1-17). ACM. <https://doi.org/10.1145/3445984>
- Scherer, R., Siddiq, F., & Sánchez Viveros, B. (2018). Learning to code - Does it help students to improve their thinking skills? In S. C. Kong, D. Andone, G. Biswas, T. Crick, H. U. Hoppe, T. C. Hsu, R. H. Huang, K. Y. Li, C. K. Looi, M. Milrad, J. Sheldon, J. L. Shih, K. F. Sin, M. Tissenbaum & J. Vahrenhold (Eds.), *Proceedings of the international conference on computational thinking education 2018 (CTE2018)* (pp. 37-40). The Education University of Hong Kong.
- Scherer, R., Siddiq, F., & Viveros, B. A. S. (2020). A meta-analysis of teaching and learning computer programming: Effective instructional approaches and conditions. *Computers in Human Behavior*, *109*, Article No. 106349. <http://doi.org/10.1016/j.chb.2020.106349>
- Schryen, G., Wagner, G., & Benlian, A. (2015). Theory of Knowledge for Literature Reviews: An epistemological model, taxonomy and empirical analysis of IS literature. In L. Borzemeski, A. Grzech, J. Swiatek & Z. Wilimowska (Eds.), *Proceedings of the thirty sixth international conference on information systems* (pp. 1-22). Springer.
- Seiter, L. (2015). Using SOLO to classify the programming responses of primary grade students. In A. Decker & K. Eiselt (Eds.), *Proceedings of the 46th ACM*

- technical symposium on computer science education (SIGCSE '15)* (pp. 540–545). ACM. <https://doi.org/10.1145/2676723.2677244>
- Seiter, L., & Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. In B. Simon, A. Clear & Q. Cutts (Eds.), *Proceedings of the 9th annual international ACM conference on international computing education research (ICER '13)* (pp. 59–66). ACM. <https://doi.org/10.1145/2493394.2493403>
- Selby, C. C. (2014). *How can the teaching of programming be used to enhance computational thinking skills?* [Doctoral dissertation, University of Southampton]. University of Southampton. <https://eprints.soton.ac.uk/366256/>
- Selby, C. C. (2015). Relationships: computational thinking, pedagogy of programming, and Bloom's taxonomy. In J. Gal-Ezer, S. Sentance & J. Vahrenhold (Eds.), *Proceedings of the workshop in primary and secondary computing education (WiPSCE '15)* (pp. 80–87). ACM. <https://doi.org/10.1145/2818314.2818315>
- Selby, C. C., & Woollard, J. (2014). *Refining an understanding of computational thinking*. University of Southampton. <https://eprints.soton.ac.uk/372410/>
- Sengupta, P., Kinnebrew, J. S., Biswas, G., & Clark, D. (2013). Integrating computational thinking With K–12 science education. *Education and Information Technologies*, 18, 351–380. <http://doi.org/10.1007/s10639-012-9240-x>
- Sengupta, P., Dickes, A., & Farris, A. (2018). Toward a phenomenology of computational thinking in STEM education. In M. Khine (Ed.), *Computational thinking in STEM: Foundations and research highlights* (pp. 49–72). Springer. [http://doi.org/10.1007/978-3-319-93566-9\\_4](http://doi.org/10.1007/978-3-319-93566-9_4)
- Settle, A., & Perkovic, L. (2010). *Computational thinking across the curriculum: A conceptual framework*. Technical Reports, Paper 13. DePaul University. <http://via.library.depaul.edu/tr/13>
- Shah, N., Lewis, C., Caires, R. (2014). Analyzing equity in collaborative learning situations: A comparative case study in elementary computer science. In J. L. Polman, E. A. Kyza, D. Kevin O'Neill, I. Tabak, W. R. Penuel, A. S. Jurow, K. O'Connor, T. Lee & L. D'Amico (Eds.), *Learning and becoming in practice: The international conference of the learning sciences (ICLS) 2014 Volume 1* (pp. 495–502). International Society of the Learning Sciences.
- Sharma, K., Papavlasopoulou, S., & Giannakos, M. (2019). Coding games and robots to enhance computational thinking: How collaboration and engagement moderate children's attitudes? *International Journal of Child-Computer Interaction*, 21, 65–76. <https://doi.org/10.1016/j.ijcci.2019.04.004>
- Shute, V. J., Sun, C., & Asbell-Clarke, J. (2017). Demystifying computational thinking. *Educational Research Review*, 22, 142–158. <https://doi.org/10.1016/j.edurev.2017.09.003>
- Sormunen, K., Juuti, K., & Lavonen, J. (2019). Maker-centered project-based learning in inclusive classes: Supporting students' active participation with teacher-directed reflective discussions. *International Journal of Science*

and *Mathematics Education*, 18, 691–712. <https://doi.org/10.1007/s10763-019-09998-9>

- Standl, B. (2017). Solving everyday challenges in a computational way of thinking. In V. Dagienė & A. Hellas (Eds.), *Informatics in schools: Focus on learning programming (ISSEP 2017). Lecture notes in computer science* (pp. 180–191). Springer. [http://doi.org/10.1007/978-3-319-71483-7\\_15](http://doi.org/10.1007/978-3-319-71483-7_15)
- Statter, D., & Armoni, M. (2020). Teaching abstraction in computer science to 7th grade students. *ACM Transactions on Computing Education*, 20(1), Article No. 8. <https://doi.org/10.1145/3372143>
- Suero Montero, C. (2018). Facilitating computational thinking through digital fabrication. In M. Joy & P. Ihantola (Eds.), *Proceedings of the 18th Koli calling international conference on computing education research (Koli calling '18)* (pp. 1–2). ACM. <https://doi.org/10.1145/3279720.3279750>
- Suomala, J. (1999). *Students' problem solving in the LEGO/Logo learning environment* [Doctoral dissertation, University of Jyväskylä]. University of Jyväskylä. <https://jyx.jyu.fi/handle/123456789/13290?show=full>
- Swanson, H., Anton, G., Bain, C., Horn, M., & Wilensky, U. (2019). Introducing and assessing computational thinking in the secondary science classroom. In S.-C. Kong & H. Abelson (Eds.), *Computational thinking education* (pp. 99–117). Springer. [https://doi.org/10.1007/978-981-13-6528-7\\_7](https://doi.org/10.1007/978-981-13-6528-7_7)
- Swidan, A., Hermans, F., & Smit, M. (2018). Programming misconceptions for school students. In L. Malmi, A. Korhonen, R. McCartney, & A. Petersen (Eds.), *Proceedings of the 2018 ACM conference on international computing education research (ICER '18)* (pp. 151–159). ACM. <https://doi.org/10.1145/3230977.3230995>
- Szabo, C., Sheard, J., Luxton-Reilly, A., Simon, Becker, B. A., & Ott, L. (2019). Fifteen years of introductory programming in schools: A global overview of K–12 initiatives. In P. Ihantola & N. Falkner (Eds.), *Proceedings of the 19th Koli calling international conference on computing education research (Koli calling '19)* (Article No. 8). ACM. <https://doi.org/10.1145/3364510.3364513>
- Talbot, M., Geldreich, K., Sommer, J., & Hubwieser, P. (2020). Re-use of programming patterns or problem solving? Representation of Scratch programs by Tgraphs to support static code analysis. In T. Brinda & M. Armoni (Eds.), *Proceedings of the 15th workshop in primary and secondary computing education (WiPSCE '20)* (Article No. 16). ACM. <https://doi.org/10.1145/3421590.3421604>
- Tan, C. W., Yu, P.-D., & Lin, L. (2019). Teaching computational thinking using mathematics gamification in computer science game tournaments. In S. C. Kong, D. Andone, G. Biswas, T. Crick, H. U. Hoppe, T. C. Hsu, R. H. Huang, K. Y. Li, C. K. Looi, M. Milrad, J. Sheldon, J. L. Shih, K. F. Sin, M. Tissenbaum & J. Vahrenhold (Eds.), *Proceedings of the 2nd computational thinking education conference 2018 (CTE2018)* (pp. 167–182). The Education University of Hong Kong.



- Tang, X., Yin, Y., Lin, Q., Hadad, R., & Zhai, X. (2020). Assessing computational thinking: A systematic review of empirical studies. *Computers & Education*, 148, Article No. 103798. <https://doi.org/10.1016/j.compedu.2019.103798>
- Taylor, R. P., Cunniff, N., & Uchiyama, M. (1986). Learning, research, and the graphical representation of programming. In S. Winkler & H. S. Stone (Eds.), *Proceedings of the fall joint computing conference, 1986 (ACM '86)* (pp. 56–63). IEEE.
- Tedre, M. (2015). *The science of computing. Shaping a discipline*. Chapman and Hall/CRC.
- Tedre, M., & Denning, P. (2016). The long quest for computational thinking. In J. Sheard & C. Suero Montero (Eds.), *Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16)* (pp. 120–129). ACM. <https://doi.org/10.1145/2999541.2999542>
- Tsan, J., Lynch, C. F., Boyer, K. E. (2018). 'Alright, what do we need?': A study of young coders' collaborative dialogue. *International Journal of Child-Computer Interaction*, 17, 61–71. <https://doi.org/10.1016/j.ijcci.2018.03.001>
- Tsan, J., Vandenberg, J., Zakaria, Z., Boulden, D. C., Lynch, C., & Wiebe, E. (2021). Collaborative dialogue and types of conflict: An analysis of pair programming interactions between upper elementary students. In M. Sherriff & L. D. Merkle (Eds.), *Proceedings of the 52nd ACM technical symposium on computer science education (SIGCSE '21)* (pp. 1184–1190). ACM. <https://doi.org/10.1145/3408877.3432406>
- Tsan, J., Vandenberg, J., Zakaria, Z., Wiggins, J. B., Webber, A. R., Bradbury, A., Lynch, C., Wiebe, E., & Boyer, K. E. (2020). A comparison of two pair programming configurations for upper elementary students. In J. Zhang & M. Sherriff (Eds.), *Proceedings of the 51st ACM technical symposium on computer science education (SIGCSE '20)* (pp. 346–352). ACM. <https://doi.org/10.1145/3328778.3366941>
- Tsarava, K., Leifheit, L., Ninaus, M., Román-González, M., Butz, M. V., Golle, J., Trautwein, U., & Moeller, K. (2019). Cognitive correlates of computational thinking: Evaluation of a blended unplugged/plugged-in course. In Q. Cutts & T. Brinda (Eds.), *Proceedings of the 14th workshop in primary and secondary computing education (WiPSCE '19)* (Article No. 24). ACM. <https://doi.org/10.1145/3361721.3361729>
- Tuhkala, A., Wagner, M.-L., Iversen, O. S., & Kärkkäinen, T. (2019). Technology comprehension – Combining computing, design, and societal reflection as a national subject. *International Journal of Child-Computer Interaction*, 20, 54–63. <https://doi.org/10.1016/j.ijcci.2019.03.004>
- Turing, A. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 230–265. <https://doi.org/10.1112/plms/s2-42.1.230>

- Twigg, S., Blair, L., & Winter, E. (2019). Using children's literature to introduce computing principles and concepts in primary schools: work in progress. In Q. Cutts & T. Brinda (Eds.), *Proceedings of the 14th workshop in primary and secondary computing education (WiPSCE '19)* (Article No. 23). ACM. <https://doi.org/10.1145/3361721.3362116>
- Van Roy, P., & Haridi, S. (2003). *Concepts, techniques, and models of computer programming*. The MIT Press.
- Vandenberg, J., Tsan, J., Boulden, D., Zakaria, Z., Lynch, C., Boyer, K. E., & Wiebe, E. (2020). Elementary students' understanding of CS terms. *ACM Transactions on Computing Education*, 20(3), Article No. 17. <https://doi.org/10.1145/3386364>
- Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23, 459-494. [https://doi.org/10.1016/S0020-7373\(85\)80054-7](https://doi.org/10.1016/S0020-7373(85)80054-7)
- Vihavainen, A., Vikberg, T., Luukkainen, M., & Pärtel, M. (2013). Scaffolding students' learning using test my code. In J. Carter (Ed.), *Proceedings of the 18th ACM conference on innovation and technology in computer science education (ITiCSE '13)* (pp. 117-122). ACM. <https://doi.org/10.1145/2462476.2462501>
- Voogt, J., Fisser, P., Good, J., Mishra, P., & Yadav, A. (2015). Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies*, 20(4), 715-728. <https://doi.org/10.1007/s10639-015-9412-6>
- Waite, J., Curzon, P., Marsh, W., & Sentance, S. (2020). Difficulties with design: The challenges of teaching design in K-5 programming. *Computers & Education*, 150, Article No. 103838. <https://doi.org/10.1016/j.compedu.2020.103838>
- Waite, J. L., Curzon, P., Marsh, W., Sentance, S., & Hadwen-Bennett, A. (2018). Abstraction in action: K-5 teachers' uses of levels of abstraction, particularly the design level, in teaching programming. *International Journal of Computer Science Education in Schools*, 2(1), 14-40. <https://doi.org/10.21585/ijcses.v2i1.23>
- Webb, H. C., & Rosson, M. B. (2013). Using scaffolded examples to teach computational thinking concepts. In T. Camp & P. Tymann (Eds.), *Proceedings of the 44th ACM technical symposium on computer science education (SIGCSE '13)* (pp. 95-100). ACM. <https://doi.org/10.1145/2445196.2445227>
- Wei, X., Lin, L., Meng, N., Tan, W., Kong, S.-C., & Kinshuk (2021). The effectiveness of partial pair programming on elementary school students' computational thinking skills and self-efficacy. *Computers & Education*. 160, Article No. 104023. <https://doi.org/10.1016/j.compedu.2020.104023>
- Weintrop, D., Coenraad, M., Palmer, J., & Franklin, D. (2019). The teacher accessibility, equity, and content (TEC) rubric for evaluating computing curricula. *ACM Transactions on Computing Education*, 20(1), Article No. 5. <https://doi.org/10.1145/3371155>

- Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2015). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, 25, 127–147. <https://doi.org/10.1007/s10956-015-9581-5>
- Weng, X. (2018). Students' attitude changes through integrating computational thinking into English dialogue learning. In S. C. Kong, D. Andone, G. Biswas, T. Crick, H. U. Hoppe, T. C. Hsu, R. H. Huang, K. Y. Li, C. K. Looi, M. Milrad, J. Sheldon, J. L. Shih, K. F. Sin, M. Tissenbaum & J. Vahrenhold (Eds.), *Proceedings of the 2nd computational thinking education conference 2018 (CTE2018)* (pp. 50–55). The Education University of Hong Kong.
- Werner, L., Denner, J., & Campe, S. (2012). The Fairy Performance Assessment: Measuring computational thinking in middle school. In L. Smith King & D. R. Musicant (Eds.), *Proceedings of the 43rd ACM technical symposium on computer science education (SIGCSE '12)* (pp. 215–220). ACM. <https://doi.org/10.1145/2157136.2157200>
- Werner, L., Denner, J., & Campe, S. (2014). Children programming games: A strategy for measuring computational learning. *ACM Transactions on Computing Education*, 14(4), Article No. 24. <https://doi.org/10.1145/2677091>
- Werner, L., Denner, J., Campe, S., & Torres, D. M. (2020). Computational sophistication of games programmed by children: A model for its measurement. *ACM Transactions on Computing Education*, 20(2), Article No. 12. <https://doi.org/10.1145/3379351>
- Whyte, R., Ainsworth, S., & Medwell, J. (2019). Designing for integrated K-5 computing and literacy through story-making activities. In R. McCartney, A. Petersen, A. Robins & A. Moskal (Eds.), *Proceedings of the 2019 ACM conference on international computing education research (ICER '19)* (pp. 167–175). ACM. <https://doi.org/10.1145/3291279.3339425>
- Williamson, B. (2016). Political computational thinking: policy networks, digital governance and 'learning to code'. *Critical Policy Studies*, 10(1), 39–58. <https://doi.org/10.1080/19460171.2015.1052003>
- 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 (ECGBL), Cork, Ireland.
- Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–35. <https://doi.org/10.1145/1118178.1118215>
- Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical transactions of the Royal Society A*, 366, 3717–3725. <https://doi.org/10.1098/rsta.2008.0118>
- Wing, J. M. (22.3.2011). A definition of computational thinking from Jeannette Wing. *Computing educational research blog*. <https://computinged.wordpress.com/2011/03/22/a-definition-of-computational-thinking-from-jeannette-wing/>

- Wu, Q., & Anderson, J. R. (1990). *Problem-solving transfer among programming languages*. Technical reports. Carnegie Mellon University. <https://apps.dtic.mil/dtic/tr/fulltext/u2/a225798.pdf>
- Wu, S., Fang, J., & Lian, S. (2018). Design a computational thinking board game based on programming elements. In S. C. Kong, D. Andone, G. Biswas, T. Crick, H. U. Hoppe, T. C. Hsu, R. H. Huang, K. Y. Li, C. K. Looi, M. Milrad, J. Sheldon, J. L. Shih, K. F. Sin, M. Tissenbaum & J. Vahrenhold (Eds.), *Proceedings of the 2nd computational thinking education conference 2018 (CTE2018)* (pp. 19–20). The Education University of Hong Kong.
- Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., Tan, A. H., Hwa, L., Li, M., & Ko, A. J. (2019). A theory of instruction for introductory programming skills. *Computer Science Education*, 29(2-3), 205–253. <https://doi.org/10.1080/08993408.2019.1565235>
- Xu, Z., Ritzhaupt, A. D., Tian, F., & Umapathy, K. (2019). Block-based versus text-based programming environments on novice student learning outcomes: a meta-analysis study. *Computer Science Education*, 29(2-3), 177–204. <https://doi.org/10.1080/08993408.2019.1565233>
- Yadav, A., Hong, H., & Stephenson, C. (2016). Computational thinking for all: Pedagogical approaches to embedding 21st century problem solving in K–12 classrooms. *TechTrends*, 60, 565–568. <https://doi.org/10.1007/s11528-016-0087-7>
- Yadav, A., Good, J., & Caeli, E. N. (2018). Computational thinking in elementary classrooms: Measuring teacher understanding of computational ideas for teaching science. *Computer Science Education*, 28(4), 371–400. <https://doi.org/10.1080/08993408.2018.1560550>
- Yin, R. K. (2012). *Case Study Research Design and Methods* (5th ed.). SAGE.
- Yu, J., & Roque, R. (2019). A review of computational toys and kits for younger children. *International Journal of Child-Computer Interaction*, 21, 17–36. <http://doi.org/10.1016/j.ijcci.2019.04.001>
- Zakaria, Z., Boulden, D., Vandenberg, J., Tsan, J., Lynch, C.F., & Wiebe, E.N. (2019). Collaborative talk across two pair-programming configurations. In K. Lund, G. Niccolai, E. Lavoué, C. Hmelo-Silver, G. Gweon, & M. Baker (Eds.), *A wide lens: Combining embodied, enactive, extended, and embedded learning in collaborative settings, 13th international conference on computer supported collaborative learning (CSCL) 2019* (pp. 224–231). International Society of the Learning Sciences. <https://repository.isls.org/handle/1/1571>
- Zapata-Cáceres, M., Martín-Barroso, E., & Román-González, M. (2020). Computational thinking test for beginners: Design and content validation. In *2020 IEEE global engineering education conference (EDUCON 2020)* (pp. 1905–1914). IEEE. <https://doi.org/10.1109/EDUCON45650.2020.9125368>
- Zhang, L., & Nouri, J. (2019). A systematic review of learning computational thinking through Scratch in K-9. *Computers & Education*, 141, Article No. 103607. <https://doi.org/10.1016/j.compedu.2019.103607>

Zur-Bargury, I., Pârv, B., & Lanzberg, D. (2013). A nationwide exam as a tool for improving a new curriculum. In J. Carter (Ed.), *Proceedings of the 18th ACM conference on innovation and technology in computer science education (ITiCSE '13)* (pp. 267–272). ACM.  
<https://doi.org/10.1145/2462476.2462479>

## APPENDICES

### Appendix A. Informed consent (in Finnish)

## Tutkimuslupa

### *Tutkimuksen aihe*

Tässä tutkimuksessa tarkastellaan peruskoulun oppilaiden ohjelmointia ja opettajan toimintaa ohjelmoinnin opettamisen aikana. Tutkimuksessa tuotetaan tieteellisesti merkityksellistä uutta tietoa, joka palvelee koulujen ohjelmoinnin opettamista. Tutkimukseen osallistuminen tarjoaa mahdollisuuden päästä kokeilemaan uudenlaista ohjelmointisovellusta tietokoneella. Tutkimus toteutetaan yhteistyössä [...] koulun 4.-luokkien kanssa osana Jyväskylän yliopiston opettajankoulutuslaitokselle tehtävää väitöskirjaa.

### *Aineistonkeruu*

Tutkimuksessa kerätään aineistoa kouluaikana niiltä oppitunneilta, joilla ohjelmointia harjoitellaan. Tutkimuksen kesto määräytyy opettajan kanssa sovittavan ohjelmoinnin opetusjakson mukaisesti.

Aineistoa kerätään havainnoimalla opettajan ja oppilaiden toimintaa, tarkastelemalla oppilaiden ohjelmointitöitä sekä teettämällä oppilailla kyselyjä ja osaamistestejä. Opettajaa ja oppilaita videoidaan ja heidän puhettaan äänitetään. Oppilaiden käyttämiltä tietokoneilta tallennetaan myös näytönkaappausvideoita. Lisäksi joitakin oppilaita haastatellaan ohjelmointityöskentelyn aikana sekä kutsutaan yksittäin tai pienissä ryhmissä erillisiin haastatteluihin.

### *Tietojen luottamuksellisuus*

Tutkimuksen aikana kerättävä aineisto käsitellään Suomen tutkimuseettisen neuvottelukunnan (TENK) ohjeiden mukaisesti (<http://www.tenk.fi/fi/htk-ohje>). Aineistoa käytetään ainoastaan tutkimustarkoituksiin ja sitä käsittelee ainoastaan tutkimusta toteuttavat henkilöt. Aineistoon ei ole pääsyä tutkimusryhmän ulkopuolisilla henkilöillä.

Tutkimuksessa kerätään tunnisteellisista tiedoista oppilaiden nimet, luokat, iät, videokuvat ja puheäännet, jotta eri aineistoissa olevat tiedot voidaan tutkimustulosten analysoinnissa yhdistää. Tutkimukseen osallistuvat henkilöt eivät tule esiintymään tutkimusjulkaisuissa tunnistettavina. Kun tutkimus on päättynyt, tutkimusaineisto hävitetään.

### *Osallistumisen vapaaehtoisuus*

Huoltajat voivat huollettaviensa puolesta antaa suostumuksensa tutkimukseen osallistumisesta vapaaehtoisesti. Tutkimukseen osallistuminen voidaan keskeyttää koska hyvänsä. Päätös keskeyttää osallistumisen tai kieltäytyminen osallistumasta ei vaikuta kieltäytyneiden oppilaiden saamaan opetukseen tai ohjaukseen.

Tutkimuksen toteuttaja:

Janne Fagerlund, projektitutkija

[...], Opettajankoulutuslaitos, Jyväskylän yliopisto

sähköposti: [...]

työpuhelin: [...]

www-sivut: [...]

Väitöstutkimuksen ohjaajat:

Päivi Häkkinen, Koulutuksen tutkimuslaitos, Jyväskylän yliopisto, [...]

Mikko Vesisenaho, Opettajankoulutuslaitos, Jyväskylän yliopisto, [...]

Jouni Viiri, Opettajankoulutuslaitos, Jyväskylän yliopisto, [...]

Oppilaan ohjelmointityöskentelyä saa taltioida tutkimusta varten

Oppilaan saa kutsua haastatteluun

Oppilaan etu- ja sukunimi:

---

Huoltajan allekirjoitus ja nimenselvennys:

---

Paikka ja aika:

---



## ORIGINAL ARTICLES

### I

#### **COMPUTATIONAL THINKING IN PROGRAMMING WITH SCRATCH IN PRIMARY SCHOOLS: A SYSTEMATIC REVIEW**

by

Fagerlund, J., Häkkinen, P., Vesisenaho, M., & Viiri, J. 2021

Computer Applications in Engineering Education, Vol. 29, No. 1, 12–28  
<https://doi.org/10.1002/cae.22255>

Reproduced with kind permission by Wiley.



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

Janne Fagerlund<sup>1</sup>  | Päivi Häkkinen<sup>2</sup>  | Mikko Vesisenaho<sup>1</sup>  | Jouni Viiri<sup>1</sup> 

<sup>1</sup>Department of Teacher Education, University of Jyväskylä, Jyväskylä, Finland

<sup>2</sup>Finnish Institute for Educational Research, University of Jyväskylä, Jyväskylä, Finland

## Correspondence

Janne Fagerlund, Department of Teacher Education, University of Jyväskylä, Ruusuapuisto, P.O. Box 35, 40014 Jyväskylä, Finland.  
Email: [janne.fagerlund@jyu.fi](mailto: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

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2020 The Authors. *Computer Applications in Engineering Education* published by Wiley Periodicals LLC

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 were 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)

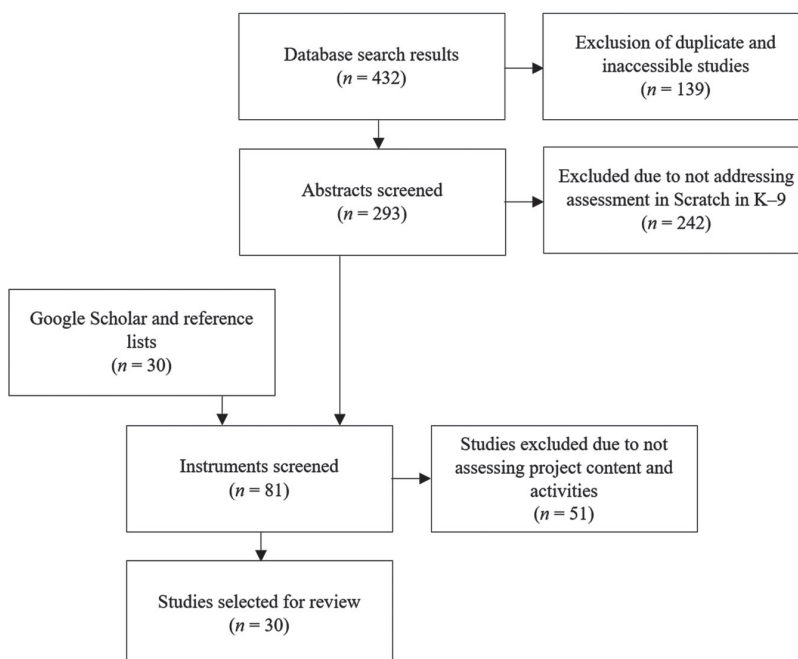


FIGURE 1 Literature search protocol



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

#	Authors	Assessment in Scratch		
		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,



1. (Uni Applying) What is the position (coordinate) of the cat after the green flag is clicked and the first instruction is run?
2. (Multi Applying) What is the position (coordinate) of the cat after the entire script is run? In what direction is the cat facing?
3. (Multi Applying) How many times is the instruction  run?
4. (Uni Creating) Add an instruction to the script so that the cat never stops.

**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 favoured 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 capsize 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	<ul style="list-style-type: none"> <li>• Sprite properties, variables and lists (abstractions of properties) [1,3,4,6,8,9,11,14,15,18,20,22,24,25,29,30]</li> <li>• Coding patterns, make-a-blocks and cloning (abstractions of behaviours) [1,5–9,11,14,20–22,24,25,28,30]</li> <li>• Continuous events (repeat until), discrete events (wait until) and initialisation (abstractions of states) [1,7,8,11,20,22,24,25,28,30]</li> <li>• Complex projects with several scripts and sprites [3,5,9,21,28]</li> </ul>
Algorithms	<ul style="list-style-type: none"> <li>• Coding patterns, make-a-blocks and cloning (coding separate procedures as specific functionalities) [1,6–9,11,14,20,22,24,25,28,30]</li> <li>• Initialisation [1,7,8,20,24,25]</li> <li>• Sequencing, looping and selection in coding patterns (algorithm control) [1,3–6,8,9,11,14,15,18,20–25,28–30]</li> <li>• Self-calling (recursive) make-a-blocks [5,22]</li> </ul>
Automation	<ul style="list-style-type: none"> <li>• 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]</li> <li>• Animations, games, art, stories and simulations (project genres) [3,9]</li> </ul>
Collaboration	<ul style="list-style-type: none"> <li>• Publishing projects [2]</li> <li>• Remixing and credit-giving [3,17]</li> <li>• Commenting, requesting friends, favouriting, “love-its” [2,3]</li> </ul>
Coordination and Parallelism	<ul style="list-style-type: none"> <li>• Synchronised parallel code constructs and coding patterns within a sprite and across sprites [3–5,9,12,18,20–25,28,29]</li> <li>• 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]</li> </ul>
Creativity	<ul style="list-style-type: none"> <li>• Customised sprites and stages [9,12,23]</li> <li>• Modifying a remixed project [3,12,29]</li> <li>• Expressing personal interest areas [3,23]</li> </ul>
Data	<ul style="list-style-type: none"> <li>• 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]</li> </ul>
Efficiency	<ul style="list-style-type: none"> <li>• Precise data manipulation [1,6,8,15,20,24,30]</li> <li>• Defined project goal [29]</li> <li>• Use instructions [9,29]</li> <li>• Functionality [9,29]</li> </ul>
Logic	<ul style="list-style-type: none"> <li>• If, if-else, nested conditionals [3,4,6,9,11,14,15,18,20–22,28–30]</li> <li>• And, or, not (Boolean logic) [3–5,9,11,14,15,18,21,28,29]</li> <li>• Arithmetic operations [3,9,15,18,29,30]</li> <li>• Absolute and relational operations [1,3]</li> </ul>
Modelling and design	<ul style="list-style-type: none"> <li>• Looks and motion animation, pen drawing and sounds (algorithm animation) [1,3–9,11,14,15,22,24,28,30]</li> <li>• No extraneous blocks [9,12,29]</li> <li>• Meaningful names for sprites and variables [9,11,12,29,30]</li> <li>• Code comments [12]</li> </ul>
Patterns and Generalisation	<ul style="list-style-type: none"> <li>• Reinstantiated code constructs [4,7,9,25]</li> <li>• Reinstantiated coding patterns [7,11,20,30]</li> </ul>
Problem decomposition	<ul style="list-style-type: none"> <li>• Coding patterns and code constructs (decomposition) [1,3–11,14–16,18,20–25,28–30]</li> <li>• Separately scripted behaviours or actions (modularisation) [3,25]</li> </ul>

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

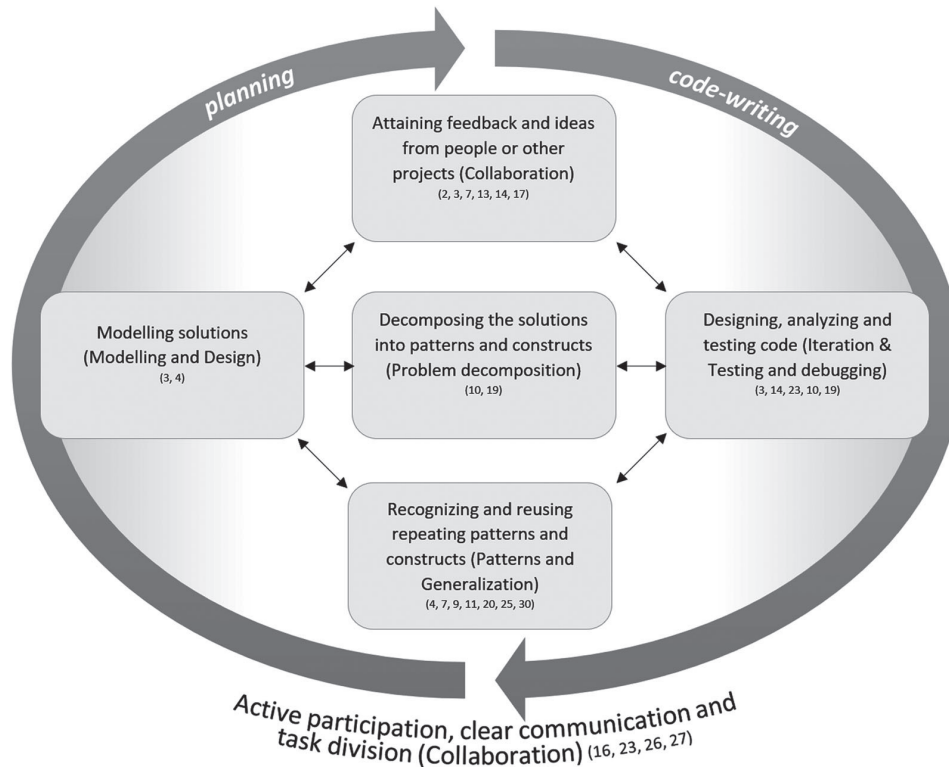


FIGURE 3 CT-fostering activities in Scratch (and source studies, see Table 1)

## 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. Chiocciariello 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*, *Comput. 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 T. M. 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-jeannette-wing/>
  63. I. Zur-Bargury, B. Pärvi 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>



## II

# ASSESSING 4<sup>TH</sup> GRADE STUDENTS' COMPUTATIONAL THINKING THROUGH SCRATCH PROGRAMMING PROJECTS

by

Fagerlund, J., Häkkinen, P., Vesisenaho, M., & Viiri, J. 2020

Informatics in Education, Vol. 19, No. 4, 611–640  
<https://doi.org/10.15388/infedu.2020.27>

Reproduced with kind permission by Vilnius University.

## Assessing 4<sup>th</sup> Grade Students' Computational Thinking through Scratch Programming Projects

Janne FAGERLUND<sup>1\*</sup>, Päivi HÄKKINEN<sup>2</sup>,  
Mikko VESISENAHO<sup>1</sup>, Jouni VIIRI<sup>1</sup>

<sup>1</sup>*Department of Teacher Education, University of Jyväskylä, Jyväskylä, Finland*

<sup>2</sup>*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 4<sup>th</sup> 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; Manilla *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 4<sup>th</sup> 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
<b>Total:</b>					<b>325</b>

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 (Grove 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<sup>1</sup>).

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

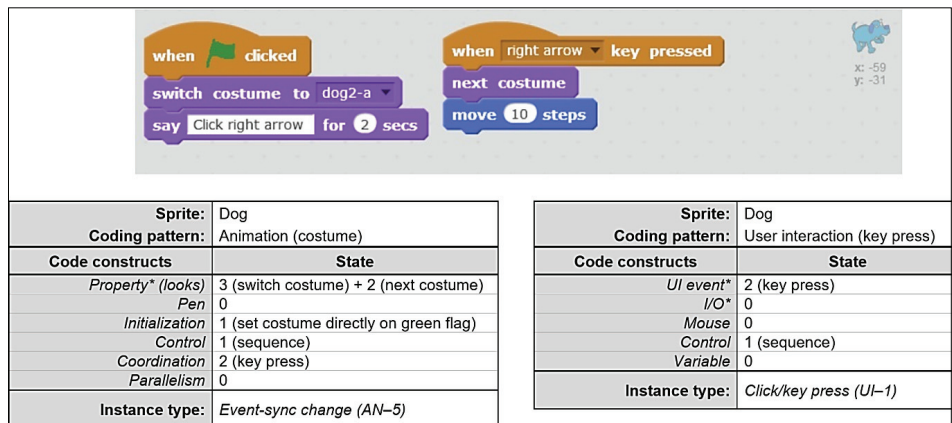


Fig. 1. Code constructs and resulting types for two example coding pattern instances.

<sup>1</sup> 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.

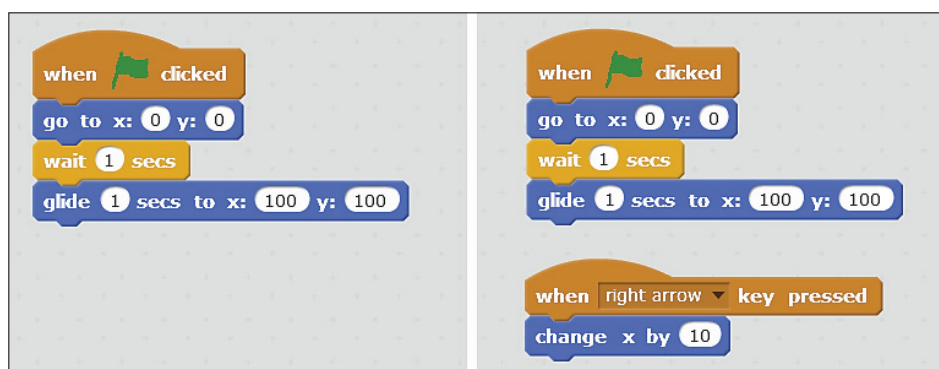


Fig. 2. Left: an instantiated “Timed animation (location)” coding pattern. Right: the location modification on the bottom (i.e., “change x by 10”) is detached as a new instance because it is coordinated with another code construct, that is, “event-sync”.

### 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.

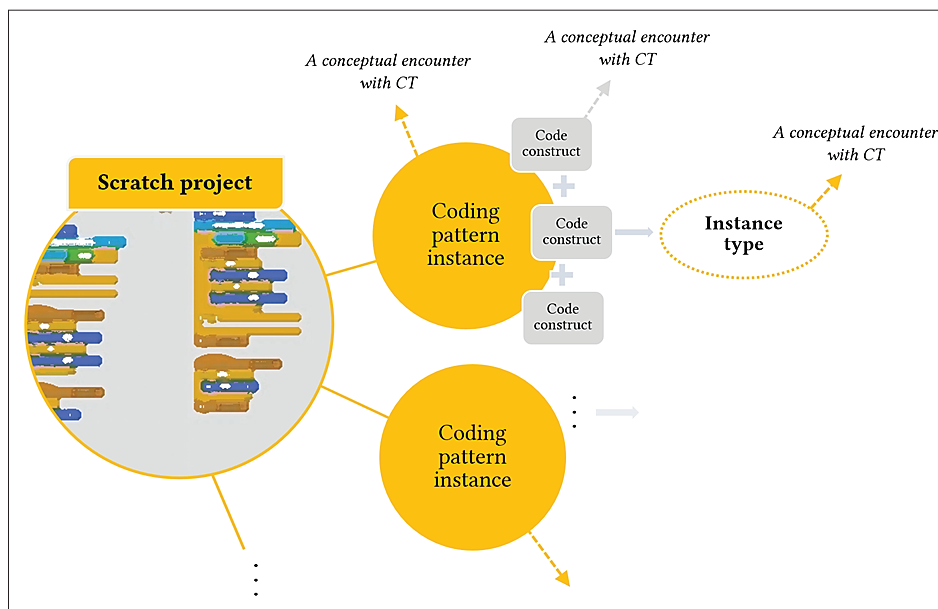


Fig. 3. Analysis of programming contents in Scratch projects in this study.

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.

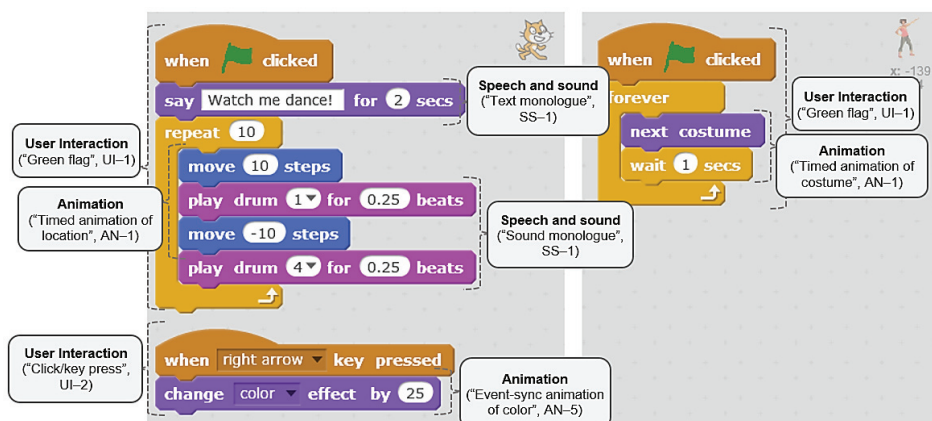


Fig. 4. A fully completed “Getting Started with Scratch” tutorial and the instances as instructed by it.

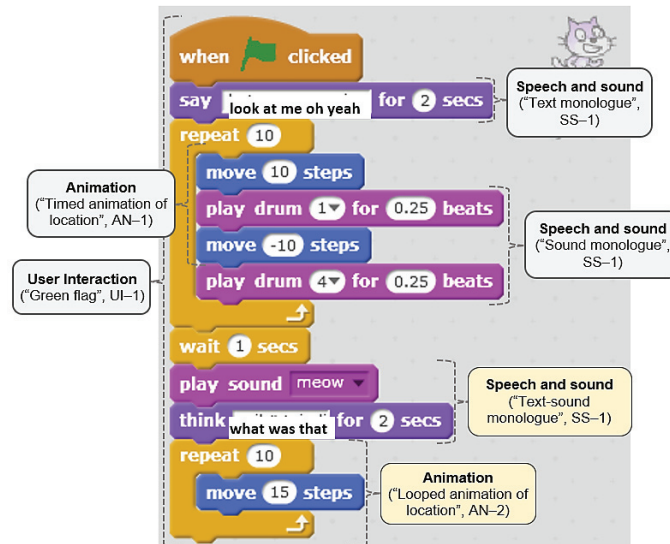


Fig. 5. A half-completed tutorial including two uninstructed instances.

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.

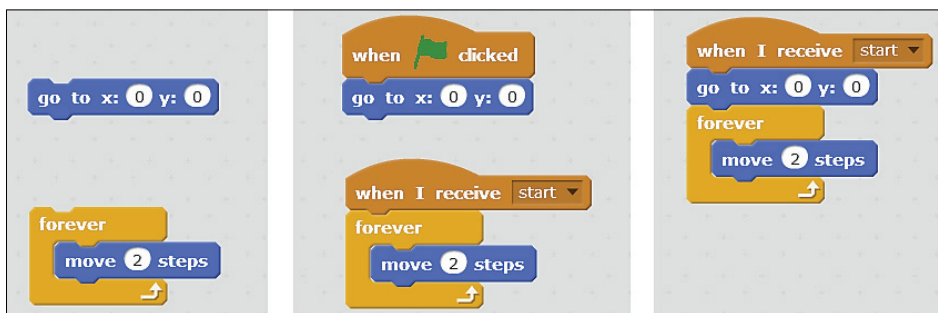


Fig. 6. Left: the initial problem. Center: “Event-sync animation (location)” (AN-5) with “initialization.” Right: “Event-sync animation (location)” (AN-5) followed by “Looped animation (location)” (AN-2) with no “initialization”.



### 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		
		<i>N</i>	Correct	Solved
<b>P4</b>				
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%
<b>P7</b>				
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” ( <i>N</i> = 33)	P3 “10 blocks” ( <i>N</i> = 22)	P8 Final projects ( <i>N</i> = 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

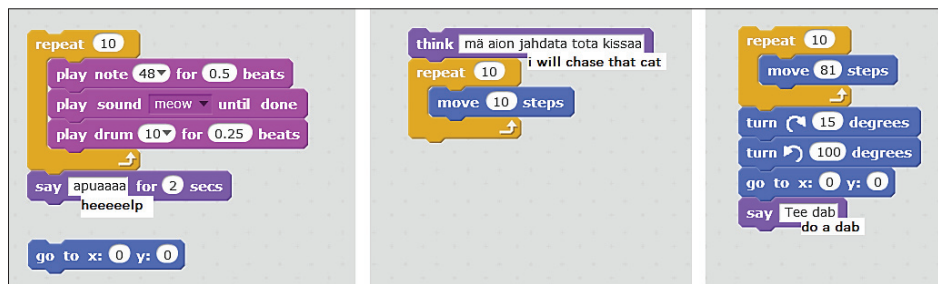


Fig. 7. Sample unscripted blocks from three P1 projects.

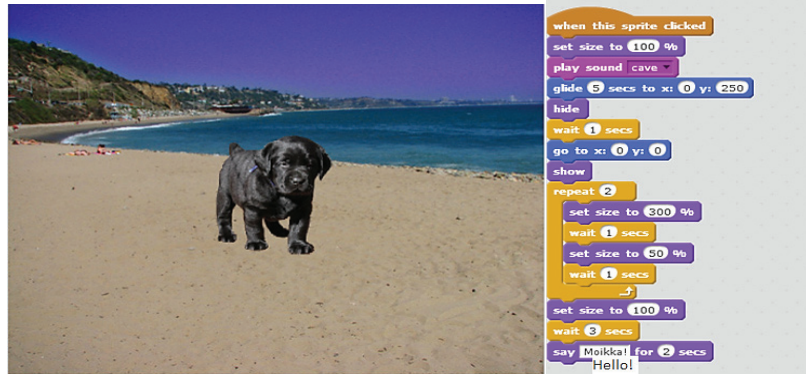


Fig. 8. An example P3 project.

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

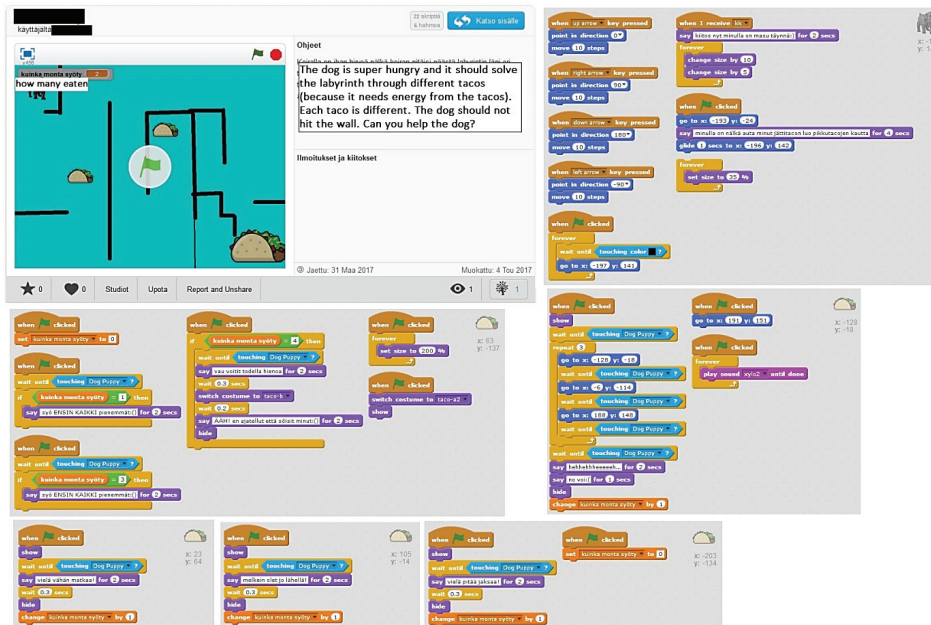


Fig. 9. The components and scripts in a relatively complex P8 project.

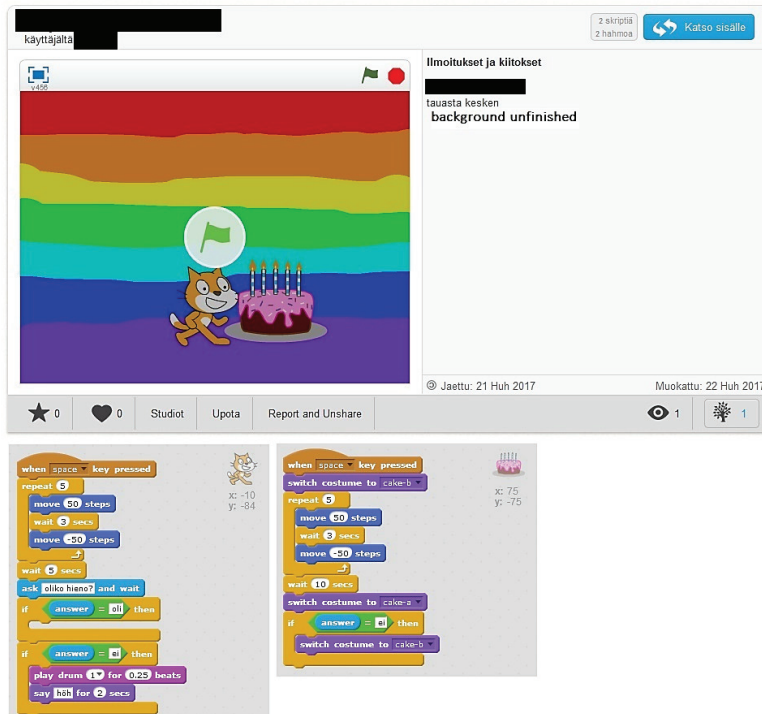


Fig. 10. The components and scripts in a relatively simple P8 project.

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
<b>Animation (AN)</b>			
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
<b>Speech and sound (SS)</b>			
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
<b>Collision (CO)</b>			
1–Test collision separately	0	5	0
2–Test collision in loop	0	5	1
3–Wait for collision	0	3	0
<b>Data manipulation (DM)</b>			
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	–
<b>User interaction (UI)</b>			
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
<b>“Control”</b>								
Number	99	8	7	4	2	15	2	30
Percentage	50%	5%	5%	4%	2%	7%	1%	4%
<b>“Coordination”</b>								
Number	113	15	16	6	3	24	6	54
Percentage	57%	10%	10%	7%	3%	11%	3%	7%
<b>“Initialization”(*)</b>								
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
<b>Abstraction</b>	Abstractions of behaviors	34%	277.4
	Abstractions of properties	63%	66.3
	Abstractions of states	54%	291.2
<b>Algorithms</b>	Algorithm control	91%	84.6
	Procedures	34%	277.4
	Starting from initial state	93%	216.6
	Recursion	0%	–
<b>Automation</b>	I/O devices	33%	152.1
<b>Coordination</b>	Coordinating scripts	57%	183.8
	Synchronizing scripts	18%	437.5
<b>Creativity</b>	Modifying remixes	81%	69.8
<b>Data</b>	Storing and manipulating data	50%	240.6
<b>Logic</b>	Boolean logic	0%	–
	Conditional structures	44%	239.0
	Operations	96%	104.5
<b>Modeling and design</b>	Algorithm animation	66%	58.6
<b>Patterns</b>	Re-instantiated coding patterns/code constructs	100%	23.1
<b>Problem decomposition</b>	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 4<sup>th</sup> 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 *et 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 instantiated 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 4<sup>th</sup> 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 Nyysönen 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., Chiocciariello, 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., Musciant, 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-jeannette-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/janefagerlund/> Postal address: Ruusupuisto, P.O. Box 35, 40014 University of Jyväskylä, Finland. E-mail: [janne.fagerlund@jyu.fi](mailto: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](mailto: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](mailto: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](mailto:jouni.p.t.viiri@jyu.fi) Tel. +358505353611





### **III**

## **FOURTH GRADE STUDENTS' COMPUTATIONAL THINKING IN PAIR PROGRAMMING WITH SCRATCH: A HOLISTIC CASE ANALYSIS**

by

Fagerlund, J., Vesisenaho, M., & Häkkinen, P.

Under review

Request a copy from author.