Informatics in Education, 2019, Vol. 18, No. 1, 17–39

DOI: 10.15388/infedu.2019.02

Approaches to Assess Computational Thinking Competences Based on Code Analysis in K-12 Education: A Systematic Mapping Study

Nathalia DA CRUZ ALVES,
Christiane GRESSE VON WANGENHEIM, Jean C.R. HAUCK
Department of Informatics and Statistics (INE) – Federal University of Santa Catarina (UFSC)
Florianópolis/SC, Brazil
e-mail: nathalia.alves@posgrad.ufsc.br, {c.wangenheim, jean.hauck}@ufsc.br

Received: September 2018

Abstract. As computing has become an integral part of our world, demand for teaching computational thinking in K-12 has increased. One of its basic competences is programming, often taught by learning activities without a predefined solution using block-based visual programming languages. Automatic assessment tools can support teachers with their assessment and grading as well as guide students throughout their learning process. Although being already widely used in higher education, it remains unclear if such approaches exist for K-12 computing education. Thus, in order to obtain an overview, we performed a systematic mapping study. We identified 14 approaches, focusing on the analysis of the code created by the students inferring computational thinking competencies related to algorithms and programming. However, an evident lack of consensus on the assessment criteria and instructional feedback indicates the need for further research to support a wide application of computing education in K-12 schools.

Keywords: assessment, code analysis, block-based visual programming language, computational thinking, K-12 education.

1. Introduction

The digital age has transformed the world and workforce, making computing and IT technologies part of our daily lives. In this context, it becomes imperative that citizens have a clear understanding of the principles and practice of computer science (CSTA, 2016). Therefore, several initiatives have emerged around the world to popularize the teaching of computing including it into K-12 education (Bocconi *et al.,* 2016). Teaching computing in school focuses on computational thinking, which refers to expressing solutions as computational steps or algorithms that can be carried out by a computer (CSTA, 2016). It involves solving problems, designing systems, and understanding human be-

havior, by drawing on the concepts fundamental to computer science (Wing, 2006). Such a competence is valuable well beyond the computing classroom, enabling students to become computationally literate and fluent to fully engage with the core concepts of computer science (Bocconi *et al.,* 2016; CSTA, 2016).

Computing is typically taught by creating, testing and refining computer programs (Shute *et al.,* 2017; CSTA, 2016; Lye and Koh, 2014; Grover and Pea, 2013). In K-12 education, block-based visual programming languages, such as Scratch (`https://scratch.mit.edu`), Blockly (`https://developers.google.com/blockly/`), BYOB/Snap! (`http://snap.berkeley.edu`) or App Inventor (`http://appinventor.mit.edu/explore/`) can be used to teach programming (Lye and Koh, 2014). Typically, programming courses include hands-on programming activities to allow students to practice and explore computing concepts as part of the learning process (Wing, 2006; Grover and Pea, 2013; Lye and Koh, 2014). This includes diverse types of programming activities, including closed and open-ended problems for which a correct solution exists (Kindborg and Scholz, 2006). Many computational thinking activities also focus on creating solutions to real-world problems, where solutions are software artifacts, such as games/animations or mobile apps (Monroy-Hernández and Resnick, 2008; Fee and Holland-Minkley, 2010). In such constructionist-based problem-based learning environments, student learning centers on complex ill-structured, open-ended problems for which no single correct answer exist (Gijselaers, 1996; Fortus *et al.,* 2004; Lye and Koh, 2014).

An essential part of the learning process is assessment and feedback (Hattie and Timperley, 2007; Shute, 2008; Black and Wiliam, 1998). Assessment guides student learning and provides feedback to both the student and the teacher (Ihantola *et al.,* 2010; Stegeman *et al.,* 2016). For effective learning, students need to know their level of performance on a task, how their own performance relates to good performance and what to do to close the gap between those (Sadler, 1989). Formative feedback, thus, consists of informing the student with the intention to modify her/his thinking or behavior for the purpose of improving learning (Shute, 2008). Summative assessment aims to provide students with information concerning what they learned and how well they mastered the course concepts (Merrill *et al.,* 1992; Keuning *et al.,* 2016). Assessment also helps teachers to determine the extent to which the learning goals are being met (Ihantola *et al.,* 2010).

Despite the many efforts aimed at dealing with the assessment of computational thinking (Grover and Pea, 2013; Grover *et al.,* 2015), so far there is no consensus nor standardization on strategies for assessing computational thinking (Brennan and Resnick, 2012; Grover *et al.,* 2014). There seems to be missing a clear definition of which assessment type to use, such as standardized multiple-choice or performance assessments based on the analysis of the code developed by the students. In this context, performance assessment seems to have a number of advantages over traditional assessments due to its capacity to assess higher-order thinking (Torrance, 1995; Ward and Lee, 2002). Thus, analyzing the student's code with respect to certain qualities may allow to assess the student's ability to program and, thus, to infer computational thinking competencies (Liang *et al.,* 2009; Moreno-León *et al.,* 2017). Yet, whereas the assessment of well-structured programming assignments with a single correct answer is straightforward (Funke, 2012), assessing complex, ill-structured problems for which no single correct solution exist is

more challenging (Eseryel *et al.,* 2013; Guindon, 1988). In this context, the assessment can be based on the assumption that certain measurable attributes can be extracted from the code, evaluating whether the students' programs show that they have learned what is expected by using rubrics. Rubrics use descriptive measures to separate levels of performance on the achievement of learning outcomes by delineating the various criteria associated with learning activities, and indicators describing each level to rate students' performance (Whittaker *et al.,* 2001; McCauley, 2003). When used in order to assess programming activities, a rubric typically maps a score to the ability of the student to develop a software artifact indirectly inferring the achievement of computational thinking competencies (Srikant and Aggarwal, 2013). Grades are determined by converting rubric scores to grades. Thus, the created outcome is assessed and a performance level for each criterion is assigned as well as a grade in order to provide instructional feedback.

Rubrics can be used manually to assess programming activities, yet being a time-consuming activity representing considerable effort (Keuning *et al.,* 2016), which may also hinder scalability of computing education (Eseryel *et al.,* 2013; Romli *et al.,* 2010; Ala-Mutka, 2005). Furthermore, due to a critical shortage of K-12 computing teachers (Grover *et al.,* 2015), many non-computing teachers introduce computing in an interdisciplinary way into their classes, facing challenges also with respect to assessment (DeLuca and Klinger, 2010; Popham, 2009; Cateté *et al.,* 2016; Bocconi *et al.,* 2016). This may further complicate the situation leaving the manual assessment error prone due to inconsistency, fatigue, or favoritism (Zen *et al.,* 2011).

In this context, the adoption of automatic assessment approaches can be beneficial easing the teacher's workload by enabling them to focus on the manual assessment of complex, subjective aspects such as creativity and/or leaving more time for other activities with students (Ala-Mutka and Järvinen, 2004). It can also help to ensure consistency and accuracy of assessment results as well as eliminate bias (Ala-Mutka, 2005; Romli *et al.,* 2010). Students can use this feedback to improve their programs and programming competencies. It can provide instant real-time instructional feedback in a continuous way throughout the learning activity, allowing them to make progress without a teacher by their side (Douce *et al.,* 2005; Koh *et al.,* 2014a; Wilcox, 2016; Yadav *et al.,* 2015). Thus, automating the assessment can be beneficial for both students and teachers, improving computing education, even more in the context of online learning and MOOCs (Vujosevic-Janicic *et al.,* 2013).

As a result, automated grading and assessment tools for programming exercises are already in use in many ways in higher education (Ala-Mutka, 2005; Ihantola *et al.,* 2010; Striewe and Goedicke, 2014). They typically involve static and/or dynamic code analysis. Static code analysis examines source code without executing the program. It is used for programming style assessment, syntax and semantics errors detection, software metrics, structural or non-structural similarity analysis, keyword detection or plagiarism detection, etc. (Truong *et al.,* 2004; Song *et al.,* 1997; Fonte *et al.,* 2013). Dynamic approaches focus on the execution of the program through a set of predefined test cases, comparing the generated output with the expected output (provided by test cases). The main aim of dynamic analysis is to uncover execution errors and to evaluate the correctness of a program (Hollingsworth, 1960; Reek, 1989; Cheang *et al.,* 2003). And, although there exist already a variety of reviews and comparisons of automated systems

for assessing programs, they are targeted on text-based programming languages (such as Java, C/C++, etc.) in the context of higher education (Ala-Mutka, 2005; Ihantola *et al.,* 2010; Romli *et al.,* 2010; Striewe and Goedicke, 2014; Keuning *et al.,* 2016).

Thus, the question that remains is which approaches exist and what are their characteristics to support the assessment and grading of computational thinking competencies, specifically on the concept of algorithms and programming, based on the analysis of code developed with block-based visual programming languages in K-12 education.

2. Related Work

Considering the importance of (automated) support for the assessment of practical programming activities in computing education, there exist several reviews of automated assessment approaches. These reviews analyze various aspects, such as feedback generation, implementation aspects as well as the impact of such tools on learning and teaching (Ala-Mutka, 2005; Ihantola *et al.,* 2010; Romli *et al.,* 2010; Caiza and Del Alamo, 2013; Striewe and Goedicke, 2014; Keuning *et al.,* 2016). Ala-Mutka (2005) presents an overview on several automatic assessment techniques and approaches, addressing the aspect on grading as well as the instructional feedback provided by the tools. Ihantola *et al.* (2010) present a systematic review of assessment tools for programming activities. The approaches are discussed from both a technical and pedagogical point of view. Romli *et al.* (2010) describe how educators taught programming in higher education as well as indicating preferences of dynamic or static analysis. Caiza and Del Alamo (2013) analyze key features related to the implementation of approaches for automatic grading, including logical architecture, deployment architecture, evaluation metrics to display on how the approach can establish a grade, and technologies used by the approaches. Striewe and Goedicke (2014) present the relation of technical results of the automated analysis in object-oriented programming with didactic benefits and the generation of feedback. Keuning *et al.* (2016) review the generation of automatic feedback. They also analyze the nature, the techniques used to generate feedback, the adaptability of tools for teachers to create activities and influence feedback, and synthesize findings about the quality and effectiveness of the assessment provided by the tools.

However, these existing reviews focus on approaches to assess code created with text-based programming languages in the context of teaching computing in higher education. Differently, the objective of this article is to provide an in-depth analysis of existing approaches for the assessment of programming activities with block-based visual programming languages in the context of K-12 education.

3. Definition and Execution of the Systematic Mapping Study

In order to elicit the state of the art on approaches to assess computer programs developed by students using block-based visual programming languages in K-12 education, we performed a systematic mapping following the procedure defined by Petersen *et al.* (2008).

3.1. *Definition of the Review Protocol*

Research Question. Which approaches exist to assess (and grade) programming activities based on code created with block-based visual programming languages in the context of K-12 education?

We refined this research question into the following analysis questions:

- **Program analysis**
  - **AQ1**: Which approaches exist and what are their characteristics?
  - **AQ2**: Which programming concepts related to computational thinking are analyzed?
  - **AQ3**: How are these programming concepts related to computational thinking analyzed?
- **Instructional feedback and assessment**
  - **AQ4**: If, and how a score is generated?
  - **AQ5**: If, and in which manner instructional feedback is presented?
- Automation of assessment
  - **AQ6**: If, and how the approach has been automated?

Data source. We examined all published English-language articles that are available on Scopus being the largest abstract and citation database of peer-reviewed literature, including publications from ACM, Elsevier, IEEE and Springer with free access through the Capes Portal[1].

Inclusion/exclusion criteria. We consider only peer-reviewed English-language articles that present an approach to the assessment of algorithms and programming concepts based on the analysis of the code. We only consider research focusing on approaches for block-based visual programming languages. And, although our primary focus is on K-12 education, we include also approaches that cover concepts commonly addressed in K-12, but which might have been used on other educational stages (such as higher education). We consider articles that have been published during the last 21 years, between January 1997 (the following year in which the first block-based programming language was created) and August 2018.

We exclude approaches that analyze code written with text-based programming languages, assess algorithms and programming concepts/practices based on other sources than the code developed by the student, such as tests, questionnaires, interviews, etc., or perform code analysis outside an educational context, e.g., software quality assessment.

Quality criteria. We consider only articles that present substantial information on the presented approach in order to enable the extraction of relevant information regarding the analysis questions.

---

[1] A web portal for access to scientific knowledge worldwide, managed by the Brazilian Ministry on Education for authorized institutions, including universities, government agencies and private companies (`www.periodicos.capes.gov.br`).

Table 1
Keywords

Core concepts	Synonyms
Code analysis	Static analysis, grading, assessment
Visual programming	Visual language, scratch, app inventor, snap, blockly

Table 2
Search string

Repository	Search string
Scopus	((“code analysis” OR “static analysis” OR grading OR assessment) AND (“visual language” OR scratch OR “app inventor” OR snap OR blockly) AND PUBYEAR > 1996)

Definition of search string: In accordance with our research objective, we define the search string by identifying core concepts considering also synonyms as indicated in Table 1. The term “code analysis” is chosen, as it expresses the main concept to be searched. The terms “static analysis”, “grading” and “assessment” are commonly used in the educational context for this kind of code analysis. The term “visual programming” is chosen to restrict the search focusing on visual programming languages. We also include the names of prominent visual programming languages used in K-12 as synonyms.

Using these keywords, the search string has been calibrated and adapted in conformance with the specific syntax of the data source as presented in Table 2.

3.2. *Execution of the Search*

The search has been executed in August 2018 by the first author and revised by the co-authors. In the first analysis stage, we quickly reviewed titles, abstracts and keywords to identify papers that matched the exclusion criteria, resulting in 36 potentially relevant articles based on the search results. In the second selection step, we analyzed the complete text of the pre-selected articles in order to check their accordance to the inclusion/exclusion criteria. All authors participated in the selection process and discussed the selection of papers until a consensus was reached. Table 3 presents the number of articles found and selected per stage of the selection process.

Table 3
Number of articles per selection stage

Initial search results	Search results analyzed	Selected after 1° stage	Selected after 2° stage
2550	2550	36	23

Many articles encountered in the searches are outside of the focus of our research question aiming at other forms of assessment such as tests (Weintrop and Wilensky, 2015) or other kinds of performance results. Several articles that analyze other issues such as, for example, the way novice students program using visual programming languages (Aivaloglou and Hermans, 2016) or common errors in Scratch programs (Techapalokul, 2017) have also been left out. Other articles have been excluded as they describe approaches to program comprehension (e.g., Zhang *et al.,* 2013; Kechao *et al.,* 2012), rather than the assessment of students' performance. We discovered a large number of articles presenting approaches for the assessment of code created with text-based programming languages (e.g. Kechao *et al.,* 2012), not considering block-based visual programming languages, which, thus, have been excluded. Articles that present an approach for evaluating other topics that do not include algorithms and programming, e.g., joyfulness and innovation of contents, were also excluded (Hwang *et al.,* 2016). Complete articles or work in progress that meet all inclusion criteria, but do not present sufficient information with respect to the analysis question have been excluded due the quality criterion (e.g., Grover *et al.,* 2016; Chen *et al.,* 2017). In total, we identified 23 relevant articles with respect to our research question (Table 4).

All relevant articles encountered in the search were published within the last nine years as shown in Fig. 1.

Table 4
Relevant articles found in the search

ID	Reference	Article title
1	(Kwon and Sohn, 2016a)	A Method for Measuring of Block-based Programming Code Quality
2	(Kwon and Sohn, 2016b)	A Framework for Measurement of Block-based Programming Language
3	(Franklin *et al.,* 2013)	Assessment of Computer Science Learning in a Scratch-Based Outreach Program
4	(Boe *et al.,* 2013)	Hairball Lint-inspired Static Analysis of Scratch Projects
5	(Moreno and Robles, 2014)	Automatic detection of bad programming habits in scratch A preliminary study
6	(Moreno-León and Robles, 2015)	Dr. Scratch – A web tool to automatically evaluate scratch projects
7	(Moreno-León *et al.,* 2016)	Comparing Computational Thinking Development Assessment Scores with Software Complexity Metrics
8	(Moreno-León *et al.,* 2017)	On the Automatic Assessment of Computational Thinking Skills – A Comparison with Human Experts
9	(Koh *et al.,* 2014a)	Real time assessment of computational thinking
10	(Koh *et al.,* 2014b)	Early validation of computational thinking pattern analysis
11	(Koh *et al.,* 2010)	Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning
12	(Basawapatna *et al.,* 2011)	Recognizing Computational Thinking Patterns
13	(Johnson, 2016)	ITCH Individual testing of computer homework for scratch assignments
14	(Seiter and Foreman, 2013)	Modeling the Learning Progressions of Computational Thinking of Primary Grade Students

Continued on next page

Table 4 - continued from previous page

ID	Reference	Article title
15	(Ota *et al.,* 2016)	Ninja code village for scratch – Function samples function analyser and automatic assessment of computational thinking concepts
16	(Werner *et al.,* 2012)	The Fairy Performance Assessment Measuring
17	(Denner *et al.,* 2012)	Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts?
18	(Wolz *et al.,* 2011)	Scrape: A tool for visualizing the code of scratch programs
19	(Maiorana *et al.,* 2015)	Quizly – A live coding assessment platform for App Inventor
20	(Ball and Garcia, 2016)	Autograding and Feedback for Snap!: A Visual Programming Language.
21	(Ball, 2017)	Autograding for Snap!
22	(Gresse von Wangenheim *et al.,* 2018)	CodeMaster – Automatic Assessment and Grading of App Inventor and Snap! Programs
23	(Funke *et al.,* 2017)	Analysis of Scratch Projects of an Introductory Programming Course for Primary School Students

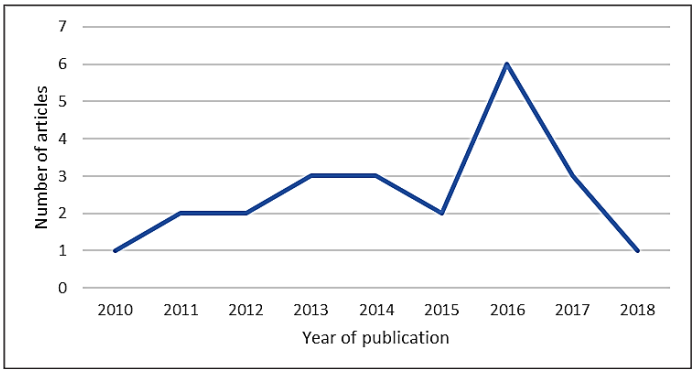

Fig. 1. Amount of publications presenting approaches on code analysis of visual programming languages in the educational context per year.

4. Data Analysis

To answer the research question, we present our findings with respect to each of the analysis questions.

4.1. *Which Approaches Exist and what are their Characteristics?*

We found 23 relevant articles describing 14 different approaches, as some of them present the same approach just from a different perspective. Most approaches have been developed to assess code created with Scratch. The approaches also differ with respect to the type of programming activity for which they are designed as shown in Table 5.

Table 5
Overview of the characteristics of the approaches encountered

ID	Name of the approach	Block-based visual programming language	Type of programming activity	
			Open-ended well-structured problem with a correct solution known in advance	Open-ended ill-structured problem without a correct solution known in advance
1 and 2	Approach by Kwon and Sohn	Visual programming language in general		x
3 and 4	Hairball	Scratch	x	
5, 6, 7 and 8	Dr. Scratch	Scratch		x
9, 10, 11 and 12	CTP/PBS/REACT	AgentSheets	x	
13	ITCH	Scratch	x	
14	PECT	Scratch		x
15	Ninja Code Village	Scratch		x
16	Fairy Assessment	Alice	x	
17	Approach by Denner, Werner and Ortiz	Stagecast Creator	x	
18	Scrape	Scratch		x
19	Quizly	App Inventor	x	
20 and 21	Lambda	Snap!	x	
22	CodeMaster	App Inventor and Snap!		x
23	Approach by Funke, Geldreich and Hubwieser	Scratch		x

4.2. *Which Programming Concepts Related to Computational Thinking are Analyzed?*

All approaches carry out a code analysis aiming at measuring the competence of programming concepts as a way of assessing computational thinking. Performing a product-oriented analysis, analyzing the code itself, the approaches look for indicators of concepts of algorithms and programming related to computational thinking practices. In accordance to the CSTA K-12 computer science framework (CSTA, 2016), most approaches analyze four of the five subconcepts related to the core concept algorithms and programming: *control*, *algorithms*, *variables* and *modularity* (Table 6). None of the approaches explores the subconcept *program development*. Thus, although, not directly measuring all the dimensions of computational thinking, these approaches intend to assess computational thinking indirectly by measuring algorithms and programming concepts through the presence of specific algorithms and program commands. Other approaches, such as CTP (Koh *et al.,* 2014b), analyze *computational thinking patterns*, including generation, collision, transportation, diffusion and hill climbing.

Some approaches also outline the manual analysis of elements related to the content of the program developed, such as *creativity* and *aesthetics* (Kwon and Sohn, 2016b; Werner *et al.,* 2012). The *functionality* of the program is analyzed only by dynamic

Table 6
Overview on the analyzed elements

Approach	Analyzed elements														
	In relation to the CSTA K-12 curriculum framework					Additional elements									
	Algorithms	Variables	Control	Modularity	Development of programs	Design technique	Creativity	Aesthetics	Functionality	Completeness level	Initialization	Computational Thinking Patterns	Function detection	Code organization and documentation	Usability design
Approach by Kwon and Sohn	x	x	x	x		x	x	x	x	x					
Hairball	x		x								x				
Dr.Scratch	x	x	x	x											
CTP/PBS/REACT	x	x	x									x			
ITCH	x	x	x	x					x						
PECT	x	x	x	x							x				
Ninja Code Village	x	x	x	x									x		
Fairy Assessment	x	x	x	x			x								
Approach by Denner, Werner and Ortiz	x	x	x	x				x	x	x				x	x
Scrape	x	x	x	x							x				
Quizly	x	x	x	x					x						
Lambda	x	x	x	x					x						
CodeMaster	x	x	x	x										x	
Approach by Funke, Geldreich and Hubwieser	x	x	x	x					x					x	x

code analysis or manual approaches. Two approaches (Kwon and Sohn, 2016b; Denner *et al.,* 2012) analyze the *completeness level* by analyzing if the program has several functions, or in case of games, several levels. Ninja Code Village (Ota *et al.,* 2016) also analyzes if there are general-purpose *functions* in the code, for example, if, in a game, a "jump" function has been implemented. Three approaches analyze code organization or documentation, e.g., meaningful naming for variables or creating procedures to organize the code (Denner *et al.,* 2012; Gresse von Wangenheim *et al.,* 2018; Funke *et al.,* 2017). Only two (manual) approaches analyze elements related to *usability* (Denner *et al.,* 2012; Funke *et al.,* 2017).

Some approaches also analyze specific competences regarding the characteristics of the programming language and/or program type, such as *computational thinking patterns* in games (Koh *et al.,* 2014b). However, just based on the code created it may not

possible to analyze fundamental practices, such as recognition and definition of computational problems (Brennan and Resnick, 2012). The assessment of other complex aspects, such as creativity is also difficult to automate, reflected by the fact that no automated approach with respect to this criterion has been encountered. To evaluate these topics other complementary forms of evaluation should be used, such as, artifact-based interviews and design scenarios (Brennan and Resnick, 2012).

4.3. How are these Programming Concepts Related to Computational Thinking Analyzed?

The approaches analyze code in different ways, including automated static or dynamic code analysis or manual code analysis. The majority of the encountered approaches uses static code analysis (Table 7). This is also related to the fact that the type of analysis depends on the type of programming activity. Only in case of open-ended well-structured problems with a solution known in advance, it is possible to compare the students' program code with representations of correct implementations for the given problem, thus allowing dynamic code analyses.

All approaches that focus on the analysis of activities with open-ended ill-structured problems are based on static code analysis, detecting the presence of command blocks. This allows identifying which and how often each command is used. In order to measure computational thinking competences, static approaches analyze the code in order to detect the presence of specific program commands/constructs inferring from their presence the learning of algorithms and programming concepts. For example, to measure competence with respect to the subconcept *control*, that specifies the order in which

Table 7
Overview on analysis types

Approach	Automated analysis		Manual analysis
	Static analysis	Dynamic analysis	
Approach by Kwon and Sohn	x		
Hairball	x		
Dr.Scratch	x		
CTP/PBS/REACT	x		
ITCH	x	x	
PECT			x
Ninja code village	x		
Fairy Assessment			x
Approach by Denner, Werner and Ortiz			x
Scrape	x		
Quizly		x	
Lambda	x	x	
CodeMaster	x		
Approach by Funke, Geldreich and Hubwieser			x

instructions are executed, they check if the student used a *loop* command in the program. This type of approach assumes that the presence of a specific command block indicates a conceptual encounter (Brennan and Resnick, 2012).

Based on the identified quantities of certain commands, further analyses are performed, for example, calculating sums, averages and percentages. The results of the analysis are presented in various forms, including charts with different degrees of detail. For example, the Scrape tool (Wolz *et al.,* 2011) presents general information about the program, the percentage of each command present in the project as well as the exact number of times each command was used per category. Some approaches present the results of the static analysis on a more abstract level beyond the quantification of commands (Moreno-León and Robles, 2015; Ota *et al.,* 2016; Gresse von Wangenheim *et al.,* 2018). An example is the Dr. Scratch tool (Moreno-León and Robles, 2015) that analyzes concepts such as abstraction, logic and parallelism providing a score for each concept based on a rubric.

Programming activities with open-ended well-structured problems can also be assessed adopting static code analysis, typically by comparing the students' code with the representation of the correct solution pre-defined by the instructor. In this case, the analysis is carried out by checking if a certain set of commands is present in the student's program (e.g., (Koh *et al.,* 2014a)). Yet, this approach requires that for each programming exercise the teacher or the instructional designer previously programs a model solution.

Some approaches adopt a dynamic code analysis (e.g., (Maiorana *et al.,* 2015)). In this case, tests are run in order to determine if the solution of the student is correct based on the output produced by the program. However, adopting this approach requires at least the predefinition of the requirements and/or test cases for the programming activity. Another disadvantage of this kind of black-box testing is that it examines the functionality of a program without analyzing its internal structure. Thus, a solution that generates a correct result may be considered correct, even when not using the intended programming constructs, e.g., repeating the same command several times instead of using a loop construct. In addition to these basic types of analysis, ITCH (Johnson, 2016) adopts a hybrid approach combining dynamic analysis (through custom tests) and static analysis for open-ended well-structured problems with a correct solution known in advance.

Several approaches rely on manual code analysis for either type of activity (with or without a solution known in advance) typically using rubrics. An example is the Fairy Assessment approach (Werner *et al.,* 2012) using a rubric to assess the code for open-ended well-structured problem. The PECT approach presents a rubric to perform manual analysis for open-ended ill-structured problems (Seiter and Foreman, 2013).

4.4. *How do the Approaches Provide Instructional Feedback?*

The assessment of the students' computational thinking competence is done by using different forms of grading. Some approaches use a dichotomous scoring attribute, assessing the correctness of a program as a whole, e.g., indicating if it is right or wrong.

An example is Quizly (Maiorana *et al.,* 2015) that tests the program of the student and, then shows a message indicating if the program is correct or incorrect as well as the error occurred.

Several approaches divide the program or competencies into areas and assign a polytomous score for each area. Therefore, a single program can receive different scores for each area (Boe *et al.,* 2013). An example is Hairball (Boe *et al.,* 2013), which labels each area as (i) correct, when the concept was implemented correctly, (ii) semantically incorrect, when the concept was implemented in a way that does not always work as intended, (iii) incorrect, when it was implemented incorrectly, or (iv) incomplete, when only a subset of the blocks needed for a concept was used.

Some approaches provide a composite score based on each of these polytomous scores. An example is Dr.Scratch (Moreno-León and Robles, 2015) that analyzes seven areas and assigns a polytomous score to each area. In this case, it is assumed that the use of blocks of greater complexity, such as, "if then, else" implies higher performance levels than using blocks of less complexity such as "if then". A final score is as-

Table 8

Overview on the assignment of scores

Approach	Type of scoring			Does not assign a score	Not identified
	Dichotomous scoring	Polytomous scoring	Composite score		
Approach by Kwon and Sohn			x (general formula)		
Hairball		x (for 4 components)			
Dr.Scratch		x (for 7 components)	x (sum of polytomous scores)		
CTP/PBS/REACT		x (for 9 components)	x (general formula)		
ITCH				x (reporting of results)	
PECT		x (for 24 multi-dimensional components)			
Ninja code village		x (for 8 components)			
Fairy Assessment			x (formula for each activity)		
Approach by Denner, Werner and Ortiz					x
Scrape				x (statistical data)	
Quizly	x				
Lambda	x				
CodeMaster		x (for 15 components)	x (sum of polytomous scores)		
Approach by Funke, Geldreich and Hubwieser		x			

signed to the student's program based on the sum of the polytomous scores. This final composite score indicates a general assessment on a 3-point ordinal scale (basic, developing, master). The composite score is also represented through a mascot, adopting gamification elements in the assessment (Moreno-León and Robles, 2015). The tool also creates a customized certificate that can be saved and printed by the student. Similar, CodeMaster (Gresse von Wangenheim *et al.,* 2018) assigns a polytomous score based on either its App Inventor or Snap! Rubric. A total score is calculated through the sum of the partial scores, and based on the total score a numerical grade and a ninja badge is assigned.

Another way of assigning a composite score is based on a weighted sum of the individual scores for each area or considering different variables. The approach presented by Kwon and Sohn (2016a) evaluates several criteria for distinct areas, each one with different weights. The CTP approach (Koh *et al.,* 2014b) assigns a total score to the program based on primitives related to computational thinking patterns.

The assessment of the approaches is intended to be used in a summative and/or formative way. Few approaches provide feedback by explicitly giving suggestions or tips on how to improve the code (Table 9).

Feedback is customized according to the results of the code analysis and involves suggestions on good practices or modifications that can be made in order to achieve a higher score and to close the gap between what is considered good performance. None of the articles report in detail how this feedback is generated. However, it can be inferred that approaches, which perform a static code analysis create this feedback based on the results of the code analysis. On the other hand, feedback given by approaches using dynamic code analysis is based on the response obtained by the execution of the program.

Table 9
Overview on types of assessment and instructional feedback

Approach	Assessment		Provides explicit suggestions or tips on how to improve the code
	Summative	Formative	
Approach by Kwon and Sohn	x		
Hairball	x	x	x
Dr.Scratch	x	x	x
CTP/PBS/REACT		x (in real time)	
ITCH	x		
PECT	x		
Ninja code village	x	x	
Fairy Assessment	x		
Approach by Denner, Werner and Ortiz Scrape	x		
Quizly		x	x
Lambda		x	
CodeMaster	x	x	
Approach by Funke, Geldreich and Hubwieser	x		

Some approaches also provide tips that can be consulted at any time. Dr.Scratch (Moreno-León and Robles, 2015) provides a generic explanation on how to achieve the highest score for each of the evaluated areas. Similar, CodeMaster (Gresse von Wangenheim *et al.,* 2018) presents the rubric used to assess the program. Quizly (Maiorana *et al.,* 2015), along with the task description, provides a link to a tutorial on how to solve exercises.

4.5. *Have the Approaches been Automated?*

Only a few approaches are automated through software tools (Table 10). Most automated approaches perform a static code analysis. Few approaches use dynamic code analysis to assess the student's solution (Johnson, 2016; Maiorana *et al.,* 2015; Ball, 2017). These software tools seem to be typically targeted at teachers and/or students, with few exceptions providing also features for administrators or institutional representatives (Maiorana *et al.,* 2015).

In general, details about the implementation of the tools are not presented in the encountered literature, with few exceptions. Hairball (Boe *et al.,* 2013) was developed as a set of scripts in Python using the object orientation paradigm, so that it can be extended and adapted to evaluate specific tasks. Dr. Scratch (Moreno-León and Robles, 2015)

Table 10
Overview on tool aspects

Approach	User categories	Access platform	License	Language
Approach by Kwon and Sohn	Teacher	Not identified	Not informed	English
Hairball	Teacher	Python scripts	Free	English
Dr.Scratch	Student, Teacher and Institution	Web application	Free	Spanish, English, Portuguese
CTP/PBS/REACT	Teacher	Web application	Not informed	English
ITCH	Student	Python scripts	Not informed	English
PECT	Teacher	Rubric (not automated)	Not informed	English
Ninja code village	Student and Teacher	Web application	Free	English, Japanese
Fairy Assessment	Teacher	Rubric (not automated)	Not informed	English
Approach by Den-ner, Werner, Ortiz	Teacher	Rubric (not automated)	Not informed	English
Scrape	Teacher	Desktop application	Free	English
Quizly	Student, Teacher and Administrator	Web application	Free	English
Lambda	Student and Teacher	Web application	Free	English
CodeMaster	Student, Teacher and Administrator	Web application	Free	English, Portuguese
Approach by Funke, Geldreich, Hubwieser	Teacher	Rubric (not automated)	Not informed	English

was implemented based on Hairball. It has been implemented in the Python language developing plug-ins from Hairball. CodeMaster separates the analysis/assessment and presentation into different modules. The backend system was implemented in Java 8, running on an Apache Tomcat 8 application server using a MySQL 5.7 database. The front-end component was implemented in the JavaScript using the Bootstrap library with an additional custom layout (Gresse von Wangenheim *et al.,* 2018).

The access to the tools is either in the form of scripts, web, or desktop application. As a web application, Dr. Scratch, for example, allows the user to simply inform the Scratch project's URL or to upload the exported Scratch file to run the assessment.

Although, it was not possible to get detailed license information on all tools, we were able to access a free implementation of several tools (Table 10). Most tools are available in English only, with few exceptions providing internationalization and localization for several languages, such as Japanese, Spanish, Portuguese, etc., thus, facilitating a widespread adoption in different countries.

5. Discussion

Considering the importance of automated support for the assessment of programming activities in order to widely implement the teaching of computing in K-12 education, so far, only very few approaches exist. Most of the approaches focus on analyzing Scratch code, being currently one of the most widely used block-based visual programming languages, popular in several countries. For other block-based visual programming languages, very few solutions have been encountered. The approaches are intended to be used for formative and/or summative assessment in computing education. We encountered approaches for different types of programming activities including open-ended well-structured problems with a pre-defined correct or best solution as well as open-ended ill-structured problems in problem-based learning contexts.

Although the majority of the approaches aims at supporting the assessment and grading process of the teacher, some tools are also intended to be used by the students directly to monitor and guide their learning progress. Examples are Dr.Scratch (Moreno-León and Robles, 2015), CodeMaster (Gresse von Wangenheim *et al.,* 2018) and CTP (Koh *et al.,* 2014a) that even provides real-time feedback during the programming activity. The approaches typically provide feedback in form of a score based on the analysis of the code, including dichotomous or polytomous scores for single areas/concepts as well as composite scores providing a general result. Few approaches provide suggestions or tips on how to improve the code (in addition to a single score) in order to guide the learning process. Only two approaches (Moreno-León and Robles, 2015; Gresse von Wangenheim *et al.,* 2018) use a kind of gamification by presenting the level of experience in a playful way with mascots.

For the assessment, the approaches use static, dynamic or manual code analysis to analyze the code created by the student. The advantage of static code analysis approaches measuring certain code qualities is that they do not require a pre-defined correct best solution, being an alternative for the assessment of ill-structured activities in problem-

based learning contexts. However, the inexistence of a pre-defined solution for such ill-structured activities limits their analysis with respect to certain qualities, not allowing the validation of the correctness of the code. Yet, on the other hand, in order to stimulate the development of higher order thinking, ill-structured problems are important in computing education. Dynamic code analysis approaches can be applied for the assessment of open-ended well-structured problems for which a pre-defined solution exist. However, a disadvantage, as they do not consider the internal code structure, is that they may consider a program correct (when generating the expected output), even when the expected programming commands were not used.

By focusing on performance-based assessments based on the analysis of the code created by the students, the approaches infer an assessment of computational thinking concepts and practices, specifically related to the concept of algorithms and programming, based on the code. This explains the strong emphasis of the analysis of programming-related competences, assessing mostly algorithm and programming sub-concepts, such as, *algorithms*, *variables*, *control* and *modularity* by the majority of the approaches as part of computational thinking competences. Additional elements such as usability, *code organization*, *documentation*, *aesthetics* or *creativity* are assessed only by manual approaches. This current limitation of this kind of assessment based exclusively on the analysis of the code, also indicates the need for alternative assessment methods, such as observation or interviews in order to be able to provide a more comprehensive assessment, especially when regarding computational thinking practices and perspectives (Brennan and Resnick, 2012). In this respect, approaches based on code analysis can be considered one means for the assessment of computational thinking that especially, when automated, free the teacher to focus on complementary assessment methods, rather than being considered the only way of assessment.

We also observed that there does not seem to exist a consensus on the concrete criteria, rating scales, scores and levels of performance among the encountered approaches. Few articles indicate the explicit use of rubrics as a basis for the assessment (Seiter and Foreman, 2013; Werner *et al.,* 2012; Moreno-León and Robles, 2015; Ota *et al.,* 2016; Gresse von Wangenheim *et al.,* 2018). This confirms the findings by Grover and Pea (2013) and Grover *et al.* (2015) that despite the many efforts aimed at dealing with the issue of computational thinking assessment, so far there is no consensus on strategies for assessing computational thinking concepts.

The approaches are designed for the context of teaching programming to novice students, mostly focusing on K-12. Some approaches are also being applied in different contexts including K-12 and higher education. However, with respect to K-12 education, none of the approaches indicates a more exact specification of the educational stage for which the approach has been developed. Yet, taking into consideration the large differences in child learning development stages in K-12 and, consequently the need for different learning objectives for different stages, as for example refined by the CSTA curriculum framework (CSTA, 2016), it seems essential to specify more clearly which educational stage is addressed and/or to provide differentiated solution for different stages.

Only some approaches are automated. Yet, some do not provide a user-friendly access, showing results only in a terminal that runs scripts, which may hinder their adop-

tion. Another factor that may hinder the widespread application of the tools in practice is their availability in English only, with only few exceptions available in a few other languages. Another shortcoming we observed is that these tools are provided as stand-alone tools not integrated into programming environments and/or course management systems in order to ease their adopting in existing educational contexts.

These results show that, although there exist some punctual solutions, there is still a considerable gap not only for automated assessment tools, but also for the conceptual definition of computational thinking assessment strategies with respect to the concept of algorithms and programming. As a result of the mapping study several research opportunities in this area can be identified, including the definition of well-defined assessment criteria for a wide variety of block-based visual programming languages especially for the assessment of open-ended ill-defined activities aiming at the development of diverse types of applications (games, animations, apps, etc.). Observing also the predominant focus on the assessment of programming concepts, the consideration of other important criteria such as creativity could be important, as computing education is considered not only to teach programming concepts, but also to contribute to the learning of 21th century skills in general. Another improvement opportunity seems to be the provision of better instructional feedback in a constructive way that effectively guides the student to improve learning. We also observed a lack of differentiation between different stages of K-12 education ranging from kindergarten to 12th grade, with significant changes of learning needs at different educational stages. Thus, another research opportunity would be the study of these changing needs and characteristics with respect to different child learning development stages. Considering also practical restrictions and a trend to MOOCs, the development and improvement of automated solutions, which allow an easy and real-time assessment and feedback to the students, could further improve the guidance of the students as well as reduce the workload of the teachers, and, thus, help to scale up the application of computing education in schools.

Threats to Validity. Systematic mappings may suffer from the common bias that positive outcomes are more likely to be published than negative ones. However, we consider that the findings of the articles have only a minor influence on this systematic mapping since we sought to characterize the approaches rather than to analyze their impact on learning. Another risk is the omission of relevant studies. In order to mitigate this risk, we carefully constructed the search string to be as inclusive as possible, considering not only core concepts but also synonyms. The risk was further mitigated by the use of multiple databases (indexed by Scopus) that cover the majority of scientific publications in the field. Threats to study selection and data extraction were mitigated by providing a detailed definition of the inclusion/exclusion criteria. We defined and documented a rigid protocol for the study selection and the selection was conducted by all co-authors together until consensus was achieved. Data extraction was hindered in some cases as the relevant information has not always been reported explicitly and, thus, in some cases had to be inferred. In these cases, the inference was made by the first author and carefully reviewed by the co-authors.

6. Conclusions

In this article, we present the state of the art on approaches to assess computer programs developed by students using block-based visual programming languages in K-12 education. We identified 23 relevant articles, describing 14 different approaches. The majority of the approaches focuses on the assessment of Scratch, Snap! or App Inventor programs, with only singular solutions for other block-based programming languages. By focusing on performance-based assessments based on the analysis of the code created by the students, the approaches infer computational thinking competencies, specifically related to the concept of algorithms and programming, using static, dynamic or manual code analysis. Most approaches analyze concepts directly related to algorithms and programming, while some approaches analyze also other topics such as design and creativity. Eight approaches have been automated in order to support the teacher, while some also provide feedback directly to the students. The approaches typically provide feedback in form of a score based on the analysis of the code, including dichotomous or polytomous scores for single areas/concepts as well as composite scores providing a general result. Only few approaches explicitly provide suggestions or tips on how to improve the code and/or use gamification elements, such as badges. As result of the analysis, a lack of consensus on the assessment criteria and instructional feedback has become evident as well as the need of such support to a wider variety of block-based programming languages. We also observed a lack of contextualization of these approaches within the educational setting, indicating for example on how the approaches can be completed by alternative assessment methods such as observations or interviews in order to provide a more comprehensive feedback covering also concepts and practices that may be difficult to be assessed automatically. These results indicate the need for further research in order to support a wide application of computing education in K-12 schools.

Acknowledgments

This work was partially supported by the Brazilian Federal Agency *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES) and by the *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq) through M.Sc. grants.

References

Aivaloglou, E., Hermans, F. (2016). How kids code and how we know: An exploratory study on the Scratch repository. In: *Proc. of the 2016 ACM Conference on International Computing Education Research*, New York, NY, USA, 53–61.

Ala-Mutka, K.M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2), 83–102.

Ala-Mutka, K.M., Järvinen, H.-M. (2004). Assessment process for programming assignments. In: *Proc. of IEEE Int. Conference on Advanced Learning Technologies*, Joensuu, Finland, 181–185.

Ball, M. (2017). Autograding for Snap!. *Hello World*, 3, 26.

Ball, M.A., Garcia, D.D. (2016). Autograding and feedback for Snap!: A visual programming language. In: *Proc. of the 47th ACM Technical Symposium on Computing Science Educatio*n, Memphis, TN, USA, 692–692.

Basawapatna, A., Koh, K.H., Repenning, A., Webb, D.C., Marshall, K.S. (2011). Recognizing computational thinking patterns. In: P*roc. of the 42nd ACM Technical Symposium on Computer Science Education*, Dallas, TX, USA, 245–250.

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

Bocconi, S., Chioccariello, A., Dettori, G., Ferrari, A., Engelhardt, K., Kampylis, P., Punie, Y. (2016). Developing computational thinking in compulsory education – Implications for policy and practice. Technical report, *European Union Scientific and Technical Research Reports*. EUR 28295 EN.

Boe, B., Hill, C., Len, M. (2013). Hairball: lint-inspired static analysis of scratch projects. In: *Proc. of the 44th ACM Technical Symposium on Computer Science Education*. Denver, CO, USA, 215–220.

Brennan, K., Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In: *Proc. of the 2012 Annual Meeting of the American Educational Research Association*, Vancouver, Canada.

Caiza, J.C., Del Alamo, J.M. (2013). Programming Assignments Automatic Grading: Review of Tools and Implementations. In: *Proc. Of the 7th International Technology, Education and Development Conference*, Valencia, Spain, 5691–5700.

Cateté, V., Snider, E., Barnes, T. (2016). Developing a Rubric for a Creative CS Principles Lab. In: *Proc. of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, Arequipa, Peru, 290–295.

Cheang, B., Kurnia, A., Lim, A., Oon, W-C. (2003). On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2), 121–131.

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

CSTA (2016). *K-12 Computer Science Framework*. `http://k12cs.org/wpcontent/uploads/2016/09/K%E2%80%9312-Computer-Science-Framework.pdf`

DeLuca, C., Klinger, D.A. (2010). Assessment literacy development: identifying gaps in teacher candidates' learning. *Assessment in Education: Principles, Policy & Practice*, 17(4), 419–438.

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(1), 240–249.

Douce, C., Livingstone, D., Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing*, 5(3), no.4.

Eseryel, D., Ifenthaler, D., Xun, G. (2013). Validation study of a method for assessing complex ill-structured problem solving by using causal representations. *Educational Technology and Research*, 61, 443–463.

Fee, S.B., Holland-Minkley, A.M. (2010). Teaching Computer Science through Problems, not Solutions. *Computer Science Education*, 2, 129–144.

Fonte, D., da Cruz, D., Gançarski, A.L., Henriques, P.R. (2013). A flexible dynamic system for automatic grading of programming exercises. In: *Proc. of the 2nd Symposium on Languages, Applications and Technologies*, Porto, Portugal, 129–144.

Fortus, D., Dershimer, C., Krajcik, J., Marx, R., Mamlok-Naaman, R. (2004). Design-based science and student learning. *Journal of Research in Science Teaching*, 41(10), 1081–1110.

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: *Proceeding of the 44th ACM technical symposium on Computer science education*, Denver, CO, USA, 371–376.

Funke, J. (2012). Complex problem solving. In: N.M. Seel (ed.), *The Encyclopedia of the Sciences of Learning*, New York: Springer; Jonassen, 3, 682–685.

Funke, A., Geldreich, K., Hubwieser, P. (2017). Analysis of Scratch projects of an introductory programming course for primary school students. In: *Proc. of IEEE Global Engineering Education Conference*, Athens, Greece, 2017.

Gijselaers, W.H. (1996). Connecting problem based practices with educational theory. In: L. Wilkerson, and W.H. Gijselaers (eds.), *Bringing Problem-Based Learning to Higher Education: Theory and Practice*. San Francisco: Jossey-Bass, 13–21.

Gresse von Wangenheim, C., Hauck, J.C.R., Demetrio, M.F., Pelle, R., Alves, N. d. C., Barbosa, H., Azevedo, L.F. (2018). CodeMaster – automatic assessment and grading of app inventor and Snap! programs. *Informatics in Education*, 2018, 17(1), 117–150.

Grover, S., Bienkowski, M., Niekrasz, J., Hauswirth, M. (2016). Assessing problem-solving process at scale. In: *Proc. of the Third ACM Conference on Learning @ Scale*. New York, NY, USA, 245–248.

Grover, S., Cooper, S., Pea, R. (2014). Assessing computational learning in K-12. In: *Proc. of the 2014 Conference on Innovation & Technology in Computer Science Education*, Uppsala, Sweden, 57–62.

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., Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Journal of Computer Science Education*, 25(2), 199–237.

Guindon, R. (1988). Software design tasks as ill-structured problems, software design as an opportunistic process. *Microelectronics and Computer Technology Corporation*, Austin, TX, USA.

Hattie, J., Timperley, H. (2007). The power of feedback. *Review of Educational Research*, 77(1), 81–112.

Hollingsworth, J. (1960). Automatic graders for programming classes. *Communications of the ACM*, 3(10), 528–529.

Hwang, G., Liang, Z., Wang, H. (2016). An online peer assessment-based programming approach to improving students' programming knowledge and skills. In: *Proc. of the Int. Conference on Educational Innovation through Technology*, Tainan, Taiwan.

Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In: *Proc. of the 10th Koli Calling International Conference on Computing Education Research*, Koli, Finland, 86–93.

Johnson, D.E. (2016). ITCH: Individual Testing of Computer Homework for Scratch assignments. In: *Proc. of the 47th ACM Technical Symposium on Computing Science Education*, Memphis, Tennessee, USA, 223–227.

Kechao, W., Tiantian, W., Xiaohong, S., Peijun, M. (2012). Overview of program comprehension. In: *Proc. of the Int. Conference Computer Science and Electronics Engineering*, London, UK, 23–25.

Keuning, H., Jeuring, J., Heeren, B. (2016). Towards a systematic review of automated feedback generation for programming exercises. In: *Proc. of the ACM Conference on Innovation and Technology in Computer Science Education*, Arequipa, Peru, 41–46.

Kindborg, M., Scholz, R. (2006). MagicWords – A Programmable Learning Toy. In: *Proc. of the Conference on Interaction Design and Children*, Tampere, Finland, 165–166

Koh, K.H., Basawapatna, A., Bennett, V., Repenning, A. (2010). Towards the automatic recognition of computational thinking. In: *Proc. of the IEEE International Symposium on Visual Languages and Human-Centric Computing*, Madrid, Spain, 59–66.

Koh, K.H., Basawapatna, A., Nickerson, H., Repenning, A. (2014a). Real time assessment of computational thinking. In: *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing*, Melbourne, Australia, 49–52.

Koh, K.H., Nickerson, H., Basawapatna, A., Repenning, A., (2014b). Early validation of computational thinking pattern analysis. In: *Proc. of the Annual Conference on Innovation and Technology in Computer Science Education*, Uppsala, Sweden, 213–218.

Kwon, K.Y., Sohn, W-S. (2016a). A method for measuring of block-based programming code quality. *International Journal of Software Engineering and its Applications*, 10(9), 205–216.

Kwon, K.Y., Sohn, W.-S. (2016b). A framework for measurement of block-based programming language. *Asia-Pacific Proc. of Applied Science and Engineering for Better Human Life*, 10, 125–128.

Liang, Y., Liu, Q., Xu, J., Wang, D. (2009). The recent development of automated programming assessment. In: *Proc. of Int. Conference on Computational Intelligence and Software Engineering*, Wuhan, China.

Lye, S.Y., and 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.

Maiorana, F., Giordano, D., Morelli, R., (2015). Quizly: A live coding assessment platform for App Inventor. In: *Proc. of IEEE Blocks and Beyond Workshop*, Atlanta, GA, USA, 25–30.

McCauley, R. (2003). Rubrics as Assessment Guides. *Newsletter ACM SIGCSE Bulletin*, 35(4), 17–18.

Moreno, J., Robles, G. (2014). Automatic detection of bad programming habits in scratch: A preliminary study. In: *Proc. of Frontiers in Education Conference*, Madrid, Spain.

Moreno-León, J., Robles, G. (2015). Dr. Scratch: a web tool to automatically evaluate Scratch projects. In: *Proc. of the 10th Workshop in Primary and Secondary Computing Education*, London, UK, 132–133.

Moreno-León, J., Robles, G., Román-González, M. (2016). Comparing computational thinking development assessment scores with software complexity metrics. In: *Proc. of IEEE Global Engineering Education Conference*, Abu Dhabi, UAE, 1040–1045.

Moreno-León, J., Román-González, M., Harteveld, C., Robles, G. (2017). On the automatic assessment of computational thinking skills: A comparison with human experts. In: *Proc. of the CHI Conference on Human Factors in Computing Systems*, Denver, CO, USA, 2788–2795.

Merrill, D.C., Reiser, B.J., Ranney, M., Trafton, J.G. (1992). Effective tutoring techniques: A comparison of human tutors and intelligent tutoring systems. *Journal of Learning Sciences*, 2(3), 277–305.

Monroy-Hernández, A., Resnick, M. (2008). Empowering kids to create and share programmable media. *Interactions*. 15(2), 50–53.

Ota, G., Morimoto, Y., Kato, H. (2016). Ninja code village for scratch: Function samples/function analyser and automatic assessment of computational thinking concepts. In: *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing*, Cambridge, UK.

Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M. (2008). Systematic mapping studies in software engineering. In: *Proc. of the 12th Int. Conference on Evaluation and Assessment in Software Engineering*, Swindon, UK, pp. 68–77.

Popham, W.J. (2009). Assessment literacy for teachers: Faddish or fundamental?. *Theory into Practice*, 48(1), 4–11.

Reek, K.A. (1989). The TRY system -or- how to avoid testing student programs. In: *Proc. of the 20th SIGCSE Technical Symposium on Computer Science Education*, 112–116.

Romli, R., Sulaiman, S., Zamli, K.Z. (2010). Automatic programming assessment and test data generation – a review on its approaches. In: *Proc. of the Int. Symposium in Information Technology*, Kuala Lumpur, Malaysia, 1186–1192.

Sadler, D.R. (1989). Formative assessment and the design of instructional systems. *Instructional Science*, 18(2), 119–144.

Seiter, L., Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. In: *Proc. of the 9th Annual Int. ACM Conference on International Computing Education Research*, San Diego, CA, USA, 59–66.

Shute. V.J. (2008). Focus on formative feedback. *Review of Educational Research*, 78(1), 153–189.

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

Srikant, S., Aggarwal, V. (2013). Automatic grading of computer programs: A machine learning approach. In: *Proc. of 12th Int. Conference on Machine Learning Applications*, Miami, FL, USA.

Song, J.S., Hahn, S.H., Tak, K.Y., Kim, J.H. (1997). An intelligent tutoring system for introductory C language course. *Computers & Education*, 28(2), 93–102.

Stegeman, M., Barendsen, E., Smetsers, S. (2016). Designing a rubric for feedback on code quality in programming courses. In: *Proc. of the 16th Koli Calling International Conference on Computing Education* Research, Koli, Finland, 160–164.

Striewe, M., Goedicke, M. (2014). A review of static analysis approaches for programming exercises. *Communications in Computer and Information Science*, 439.

Techapalokul, P. (2017). Sniffing through millions of blocks for bad smells. In: *Proc. of the ACM SIGCSE Technical Symposium on Computer Science Education*, NY, USA.

Torrance, H. (1995). *Evaluating authentic assessment: Problems and possibilities in new approaches to assessment*. Buckingham: Open University Press.

Truong, N., Roe, P., Bancroft, P. (2004). Static analysis of students' Java programs. In: *Proc. of the 6th Australasian Computing Education Conference*, Dunedin, New Zealand.

Vujosevic-Janicic, M., Nikolic, M., Tosic, D., Kuncak, V. (2013). On software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology*, 55(6), 1004–1016.

Ward, J.D., Lee, C.L. (2002). A review of problem-based learning. *Journal of Family and Consumer Sciences Education*, 20(1), 16–26.

Weintrop, D., Wilensky, U. (2015). To block or not to block, that is the question: Students' perceptions of blocks-based programming. In: P*roc. of the 14th Int. Conference on Interaction Design and Children*, Boston, MA, USA, 199–208.

Werner, L., Denner, J., Campe, S., Kawamoto, D.C., (2012). The Fairy performance assessment: Measuring computational thinking in middle school. In: *Proc. of ACM Technical Symposium on Computer Science Education*, Raleigh, NC, USA, 215–220.

Whittaker, C.R., Salend, S.J., and Duhaney, D. (2001). Creating instructional rubrics for inclusive classrooms. *Teaching Exceptional Children*, 34(2), 8–13.

Wilcox, C. (2016). Testing strategies for the automated grading of student programs. In: *Proc. of the 47th ACM Technical Symposium on Computing Science Education*, Memphis, TE, USA, 437–442.

Wing, J. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–36.

Wolz, U., Hallberg, C., Taylor, B. (2011). Scrape: A tool for visualizing the code of scratch programs. In: *Proc. of the 42nd ACM Technical Symposium on Computer Science Education*, Dallas, TX, USA.

Yadav, A., Burkhart, D., Moix, D., Snow, E., Bandaru, P., Clayborn, L. (2015). Sowing the seeds: A landscape study on assessment in secondary computer science education. In: *Proc. of the CSTA Annual Conference*, Grapevine, TX, USA.

Zhang, Y., Surisetty, S., Scaffidi, C. (2013). Assisting comprehension of animation programs through interactive code visualization. *Journal of Visual Languages & Computing*. 24(5), 313–326.

Zen, K., Iskandar, D.N.F.A., Linang, O. (2011). Using Latent Semantic Analysis for automated grading programming assignments. In: *Proc. of the Int. Conference on Semantic Technology and Information Retrieval*, Kuala Lumpur, Malaysia, 82 –88.

N. da Cruz Alves is a master student of the Graduate Program in Computer Science (PPGCC) at the Federal University of Santa Catarina (UFSC) and a research student at the initiative Computing at Schools/INCoD/INE/UFSC.

C. Gresse von Wangenheim, is a professor at the Department of Informatics and Statistics (INE) of the Federal University of Santa Catarina (UFSC), Florianópolis, Brazil, where she coordinates the Software Quality Group (GQS) focusing on scientific research, development and transfer of software engineering models, methods and tools and software engineering education in order to support the improvement of software quality and productivity. She also coordinates the initiative Computing at Schools, which aims at bringing computing education to schools in Brazil. She received the Dipl.-Inform. and Dr. rer. nat. degrees in Computer Science from the Technical University of Kaiserslautern (Germany), and the Dr. Eng. degree in Production Engineering from the Federal University of Santa Catarina. She is also PMP – Project Management Professional and MPS. BR Assessor and Implementor.

J.C.R. Hauck holds a PhD in Knowledge Engineering and a Master's Degree in Computer Science from the Federal University of Santa Catarina (UFSC) and a degree in Computer Science from the University of Vale do Itajaí (UNIVALI). He held several specialization courses in Software Engineering at Unisul, Univali, Uniplac, Uniasselvi, Sociesc and Uniarp. He was a visiting researcher at the Regulated Software Research Center – Dundalk Institute of Technology – Ireland. He is currently a Professor in the Department of Informatics and Statistics at the Federal University of Santa Catarina.