A method for assessing computational thinking in students using source code analysis

Steven Pacheco-Portuguez*, Antonio Gonzalez-Torres†, Lilliana Sancho-Chavarria*
Ignacio Trejos-Zelaya*, Jorge Monge-Fallas‡, Jose Navas-Su*
Alberto J. Cañas§, Andrés Rodríguez§, Carol Angulo Chinchilla§
* Dept. of Computing, Costa Rica Institute of Technology, Cartago, Costa Rica
† Dept. of Computer Engineering, Costa Rica Institute of Technology, Cartago, Costa Rica
‡ Dept. of Mathematics, Costa Rica Institute of Technology, Cartago, Costa Rica
§ Fundación Omar Dengo, Costa Rica
{stpacheco}@ic-itcr.ac.cr, {antonio.gonzalez, lsancho, itrejos, jomonge, jnavas}@tec.ac.cr
{alberto.canas, andres.rodriguez, carol.angulo}@fod.ac.cr

***Abstract*—This paper summarizes research in progress, at a national scale, that analyzes large volumes of elementary school and high school projects to assess the development of skills, attitudes, and practices that students develop when solving problems via computer programming, grounded on their understanding of fundamental concepts in computing. The approach is independent of programming languages and uses a generic abstract syntax tree and projects' metadata to calculate metrics, and establish relationships between measures, school regions, educational centers, and student groups. The research relates source code analysis using abstract syntax trees to some of the main computational thinking concepts. Preliminary results obtained using the proposed method are presented and discussed.**

***Index Terms*—Code analysis, computational thinking assessment, generic abstract syntax tree**

I. Introduction

Computational Thinking (CT) is a high-level competency that encourages critical thinking, creativity, communication, and cooperation among students [1]. It helps the development of abstract-mathematical, pragmatic-engineering thinking and problem solving [2]. There is still no consensus on the definition of CT [3]. PRONIE's [1] definition states that CT is the set of skills, attitudes and practices that students develop when solving problems through computer programming, grounded on their understanding of fundamental concepts in computing.

The evaluation of CT is of great interest to the education research community [4]. Source code analysis can aid appraise students' CT via the problems they solve [5]. The automatic assessment of CT could require specific analysis methods for each language and an individual analyzer for all present and future languages. An alternative is to rely upon a *single* analyzer based on a *generic* representation for all programming languages.

This research uses a Generic Abstract Syntax Tree (GAST) that accommodates multiple programming languages [6]. A common structure enables the implementation of a single analyzer for several programming languages to determine the CT level of students from grades 1 to 9. It aims to contribute in automatically performing assessments for each educational level, based on the students' programming solutions.

The method assesses CT based on the analysis of programs in diverse languages via metrics, source code structuring, data representation, abstraction, problem decomposition, control flow, and common programming errors. The data used is a subset of the exercises solved by nearly $800,000$ K-12 students in $1,281$ educational centers in Costa Rica.

The remainder of the article reviews some related work (Section II), discusses the design of the method (Section III), analyzes the results (Section IV) and discusses the conclusions and future work (Section V).

II. Related work

The evaluation of CT usually requires analyzing students' solutions to problems, which involves the inspection of structures and source code. Maloney et al. [7] performed the analysis of 536 Scratch projects of 8 to 18 year old kids in a Computer Clubhouse to track skills, compare programming concepts such as loops, boolean logic, variables, and random numbers. Likewise, Dr. Scratch [8] infers competencies such as abstraction and decomposition, logical thinking, parallelism and flow control. These inferences provide a CT score in a range of 0 to 21 points to reveal the performance of students and provide feedback for them and their teachers.

Hairball [9] is a Scratch automated static code analysis tool that provides plugins to design diverse metrics. It analyzes *say* and *sound* blocks results, synchronization between *broadcast* and *receives* blocks, and time and cycles for complex animations. There are other research works based on the creation of ASTs as intermediate representations before code analysis. SAT [10], for example, uses ANTLR to specify a grammar, and build the AST of Scratch for performing faster analyses than those of Hairball. The scores evaluate concepts such as abstraction and decomposition, parallelism, logical thinking, user interactivity, and synchronization.

Another static code analysis tool is LitterBox, which is an open-source Java project [11]. Its main features include the

[1]The PRONIE MEP-FOD program, PRONIE for short, is a joint effort between the Ministry of Public Education of Costa Rica and the Omar Dengo Foundation (FOD) for teaching computing skills in Costa Rica.

capability to identify code anomalies ('smells', bugs, 'perfumes'), calculate some metrics and perform code translation. It obtains the AST of Scratch projects and uses a Visitor pattern to find code issues, bug patterns and code smells.

III. Analysis of Computational Thinking

PRONIE uses GECO, a system to store and publish the source code of the solutions developed by students. It is independent of the programming language and also stores metadata about projects and students. So far, the GAST used in this research supports the representation of the language-specific ASTs for Java, C#, Python, Scratch and RPG to aid source code analysis [6]. Figure 1 shows the context of the engine that transforms the source code of student solutions into the GAST. It performs the following actions:

1) Extract the source code of student programs developed in Java, Alice, Arduino, Scratch, or Python.
2) Parse the source code of the solutions.
3) Create a specific AST for each student solution, according to the associated programming language.
4) Map the attributes of each AST to the corresponding ones in the GAST.
5) Analyze the GAST attributes for each student solution.
6) Generate a visualization of the analysis results.

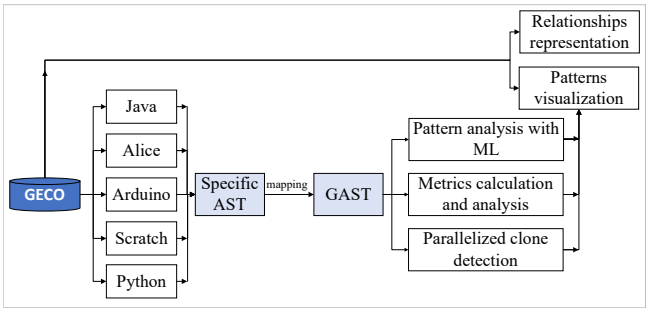

Fig. 1: Transformation and analysis engine.

The assessments reported in this paper focused on K-12 level students and Scratch programs based on (see Figure 2):

Analysis of parallelism, data representation, abstraction and decomposition (A&D), and flow control.

Common programming errors such as dead code, duplicated code, inadequate variables naming, and functions.

Code structure issues on control structures and operators.

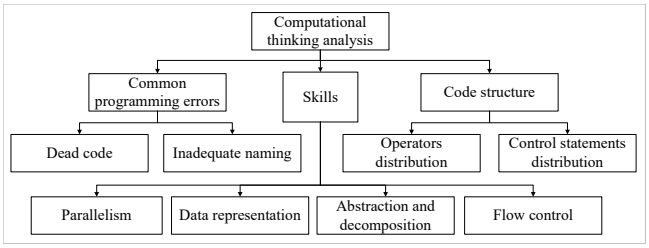

Fig. 2: Analysis of CT based on source code.

The assessment uses the rubric[2] in Table I to score the use of parallelism, data representation, abstraction, problem decomposition, and control flow [4]. The grading of skills ranges from 0 to 3, where 0 means an absence of that skill, 1 grades it as basic, 2 corresponds to an intermediate level, and 3 to a high level [8]. The maximum score for a problem solution is 12 points.

The detection of code smells and their correction reduces the cognitive load of locating potential errors [12]. Three types of common programming errors found among students are:

Dead code: Code not used or executed by any event.
Duplicated code: Stacks of code with the same instructions.
Inadequate naming: Inadequate naming of variables.

The goal of detecting these programming errors is to avoid the presence of distractors that confuse students and hinder their learning process. Eliminating these errors can increase the abstraction capabilities of students by enabling them to interpret real world problems.

The analysis of code structure in terms of control flow and operators enables the identification of the complexity of solving a problem and whether the solution fulfills the learning objectives of specific programming concepts. The outcome of those analyses are statistics and classifications according to their scope, which also enable various visualizations.

IV. Results and Analysis

The data analyzed is a private dataset from PRONIE students and is composed of $10,338$ Scratch version 2.0 projects produced between years 2018 and 2020. The metadata associated to the projects includes the school name, name and date of the project, name of the student group, and geographic details of the educational center.

The analysis demonstrated that the GAST structure enables the evaluation of source code, the calculation of CT metrics, the detection of common programming errors, and insights on the distribution of instructions in code. The method has the ability to analyze projects in Java, Python, C# and Scratch versions 2.0 and 3.0, though the results presented herein are based on Scratch version 2.0 projects due to data availability.

The dataset corresponds to the projects "New opportunities for physical computing" ($P1$) and "How can a natural phenomenon turn into a disaster?"($P2$) from 4th-grade elementary students (primary) in all country regions, with a sample of 5098 and 1238 Scratch projects for $P1$ and $P2$, respectively. The results in this paper do not include code smells nor flow control metrics. However, the preliminary results help to observe the abilities and distributions of the use of structures.

Table II reveals that for $P1$ the average value of data representation is lower than those for the other skills. The foregoing indicates that some students who developed these projects did not use variables or lists frequently. Also, the results show that project $P2$ presents a lower level of computational thinking skills than $P1$, where students exhibit a lower score in Parallelism and A&D.

[2]A midterm goal is to develop a comprehesive rubric to track the progress – in time – of the CT proficiency of students.

TABLE I: Computational Thinking Assessment Rubric

CT Skill	Null (0pt)	Basic (1pt)	Intermediate (2pt)	Proficient (3pt)
Parallelism	Does not present repetition of events	An event appears two times with different instructions	An event appears three or more times, or uses two different events that repeat two times with different instructions	Two or more events appear three or more times with different instructions.
Abstraction and problem decomposition	Uses only one code stack	Uses two code stacks in one sprite or stage	Uses two or more sprites and at least one has two or more stacks	Uses two or more sprites but at least one defines their own functions and uses them
Data representation	Does not create or use variables	Creates and uses one variable	Creates two or more variables and uses them	Creates lists and uses them
Flow control	Cyclomatic complexity is zero	Cyclomatic complexity is two	Cyclomatic complexity is three	Cyclomatic complexity is equal or greater to four

TABLE II: Results of 4th-grade students of elementary (primary) schools of all country regions.

	Statement			Operator			Skill			
Project	**If**	**While**	**For**	**Arithmetic**	**Logical**	**Comparision**	**A&D**	**Parallelism**	**Data Representation**	**Score**
New opportunities in physical computing (5098)	6.966	5.240	2.627	0.076	0.008	3.306	1.911	1.891	1.708	5.511
How can a natural phenomenon turn into a disaster? (1238)	10.707	6.660	1.359	0.105	3.352	11.696	1.349	1.311	1.417	4.079

The analysis of operators usage shows that $P2$ students use the comparison and logical operators in a higher degree than students from $P1$. The difference between both projects is minor. This gives clues that, in general, the students do not use this type of operators in the construction of their solutions. The use of control structures is more frequent in $P2$ than in $P1$, which agrees with the use of comparison operators in program control flow.

The results of $P2$ show that the programs are structurally more complex than the ones of $P1$. There is also a lower presence of proficiency, which explains the result of a lower level of data representation.

These preliminary results, based on the automated code analysis of solutions provided by students and the assessment of CT, can aid decision making concerning the achievement of learning objectives. The results can also support the identification of students with difficulties in developing specific skills, so as to focus on them and support their learning process. Furthermore, the results can help in selecting teachers whose students have higher scores of CT and appoint them to share their teaching experiences, pedagogy, and tools.

V. Conclusions and future work

Assessing the progress of the proficiency of CT in students makes it possible to improve teaching systems and reduce the digital divide. The results reported herein show that the proposed method enables the assessment of different CT skills to know the control flow structure, data representations, application of abstraction, problem decomposition, and the use of operators for a given program.

The results reveal that the GAST enables the reuse of metrics to evaluate the structural composition of programs, which proved to be helpful in evaluating CT. The results presented here are research in development, with ongoing work towards the discovery of patterns via clustering and recognition techniques to identify student profiles according to their progress.

References

[1] T. Doleck, P. Bazelais, D. J. Lemay, A. Saxena, and R. B. Basnet, "Algorithmic thinking, cooperativity, creativity, critical thinking, and problem solving: exploring the relationship between computational thinking skills and academic performance," *Journal of Computers in Education*, vol. 4, no. 4, pp. 355–369, Dec. 2017.

[2] M. Álvarez Rodríguez, "Desenvolupament del pensament computacional en educació primària: una experiència educativa amb scratch," *Revista de Ciències de l'Educació*, vol. 1, no. 2, pp. 45–64, 2017.

[3] X. Tang, Y. Yin, Q. Lin, R. Hadad, and X. Zhai, "Assessing computational thinking: A systematic review of empirical studies," *Computers & Education*, vol. 148, p. 103798, 2020.

[4] Y. Allsop, "Assessing computational thinking process using a multiple evaluation approach," *International Journal of Child-computer Interaction*, vol. 19, pp. 30–55, 2019.

[5] A. Gonzalez-Torres, L. Sancho-Chavarria, M. Zuniga-Cespedes, J. Monge-Fallas, and J. Navas-Su, "A strategy to assess computational thinking," in *8th Annual Conf. on Computational Science & Computational Intelligence (CSCI'21)*, 2021.

[6] J. Leitón-Jiménez and L. A. Barboza-Artavia, "Método para convertir código fuente escrito en diversos lenguajes de programación a un lenguaje universal," Master's thesis, Costa Rica Institute of Technology, oct 2021.

[7] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, "Programming by choice: urban youth learning programming with Scratch," in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 2008, pp. 367–371.

[8] J. Moreno-León and G. Robles, "Dr. Scratch: A web tool to automatically evaluate Scratch projects," in *Proceedings of the Workshop in Primary and Secondary Computing Education*, 2015, pp. 132–133.

[9] B. Boe, C. Hill, M. Len, G. Dreschler, P. Conrad, and D. Franklin, "Hairball: Lint-inspired static analysis of Scratch projects," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 2013, pp. 215–220.

[10] 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," in *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*. IEEE, 2018, pp. 950–955.

[11] G. Fraser, U. Heuer, N. Körber, E. Wasmeier *et al.*, "Litterbox: A Linter for Scratch programs," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 2021, pp. 183–188.

[12] G. A. Campbell, "Cognitive complexity — an overview and evaluation," in *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2018, pp. 57–58.