Assessment of Computational Thinking Skills

Submitted by:
Kevin Yeong Yu Heng
A0143487X

In partial fulfillment of the
requirements for the Degree of
Bachelor of Engineering (Computer Engineering)
National University of Singapore

B. Eng Dissertation

Assessment of Computational Thinking Skills

Submitted by:
Kevin Yeong Yu Heng
A0143487X

Project ID: H0981080

Project Supervisor: Dr. Bimlesh Wadhwa

ABSTRACT

In recent years, many governments and institutions have placed a greater emphasis on integrating computational thinking (CT) to the formal and informal K-12 education curriculum. These K-12 classrooms frequently make use of block-based programming environments for teaching computational thinking.

Even though there are many choices in block-based languages and coding tools, there is still a lack of CT assessment frameworks and tools for both learners and educators.

The project investigates the existing works regarding related topics. By consolidating the strategies of related works, an assessment framework and tool was designed with the intention of facilitating the learning and teaching of computational thinking.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Bimlesh Wadhwa for her patience, understanding and continual support throughout the process in my Final Year Project. I would like to thank her for the opportunity to participate in this project with the intention of improving the pedagogical aspect of assessment of computational thinking

LIST OF FIGURES

LIST OF ABBREVIATIONS

CT Computational Thinking

DEFINITION OF TERMS

K-12 Kindergarten through twelve grade (K-12, n.d.)

CONTENTS

INTRODUCTION

Computational thinking is a problem solving process involving in a methodical, logic-driven approach in understanding, analysing and deconstructing problems, thereafter, formulating solutions through the development of algorithms which take computational steps (Aho, 2011; Lee, 2016) and can be carried out by information-processing agents (Cuny, Snyder and Wing, 2010)

With the advent of computer technology in the 21st century and computing applications being extensively incorporated in every part of the world's economy and society, computational thinking has become more established and recognized as an important aspect of cognition and a complementary field to the established STEM education.

Although the science of computational thinking is primarily taught on the university level, however, in recent years, it has permeated into the formal and informal K-12 education curriculum especially when governments, education institutions and the industry recognize the importance of computational thinking in the modern world.

The computational thinking curriculum in the K-12 group currently employs a wide range of approaches in the learning and teaching process. Many methods for the teaching of computational thinking exists. Also there is a large variety of programming

languages, software and hardware can be used. Many informal curricula employ the use of programmable microcontrollers or robotics kits in their programme. For K-12 computational thinking education is conducted using visual block-based programming languages.

In many countries, the computational thinking programme is carried out in addition to formal education and as a complement to the conventional academic subjects. In many schools, computational thinking education is conducted in the form of short-term workshops or courses.

For educators to be able to effectively monitor the students' ability in computational thinking, there must be an assessment framework to allow educators to monitor their students' progress through indicators of their computational thinking skills.

With this as a basis of investigating the topic on the current computational thinking curriculum for the K-12 group, the project seeks to contribute in the development of assessment tools for learners and educators with goals of making the computational thinking curriculum more effective.

LITERATURE REVIEW

2.1 Definition of Computational Thinking

The concept of computational thinking originated in the nineteenth century (Kong, Abelson, & Ming, 2019) but it gained traction in the late twentieth century where computer use became more widespread.

The modern definition of computational thinking is commonly described as a problem solving process involving in a methodical, logic-driven approach in understanding, analysing and deconstructing problems, and designing solutions by algorithms which take computational steps (Aho, 2011) and can be carried out by information-processing means (Wing, 2010).

2.2 Background on Computational Thinking

Prior to the 1960s, the legitimacy of computer science being a field of science was highly debated, facing much scepticism by some of the scientific community who only thought of the use of computers as a tool (Zink, 2002). Pioneering the integration of computer science into the formal education occurred in 1962 when Purdue founded the first computer science department (Purdue University, n.d.). Subsequently the field of

computer science has been highly accredited and adopted by majority of higher education institutions (Denning & Tedre, 2019).

In the field of computer science, computational thinking concepts are required when understanding computational problems and devising solutions for them. Wing states that “computational thinking includes a range of mental tools that reflect the breadth of the field of computer science.” (Wing, 2006). Initially, computational thinking education was done only at a formal higher education level but ultimately emerged in the K-12 group and in more informal education.

The emergence of computational thinking education in the K-12 group and informal settings originated from the works of Seymour Papert starting in the 1960s. Papert envisioned that children can utilize computer technology in helping them in their learning and developing their thinking ability (Papert, 1971). Papert’s work is often attributed to be the precursor to the modern-day computational thinking education and more specifically in K-12 group.

Jeanette Wing expanded on the Papert’s perspective, proposing that CT is an essential skill for everyone and should be integrated as part of general children education (Wing, 2006). Even though Wing’s essay reignited interest and discussion on the topic, even before this, substantial effort has been made by government agencies, scientific

communities and other organizations to promote and adopt computational thinking into the K-12 curriculum (National Research Council, 2012).

2.3 Computational Thinking Education

In the late twentieth century, formal computational thinking education was almost exclusively found in university level curriculum whereas in the K-12 group, there were virtually no computer courses for children which focus on computational thinking aspects (Denning & Tedre, 2019). Approaching the twenty first century, access to computers and with the mainstream integration of information technology, computer science became more recognised, leading to an expansion of computer thinking education. An increase in funding also allowed more development in the computational thinking curriculum in both formal and informal education sectors.

Aside from teaching computational thinking in the conventional school curriculum, many governments and industry leaders have implemented policies to expose students to computational thinking. These policies include funding and research in computational thinking programmes for children (Williamson, 2016; National Science Foundation, 2007), implementing national education policies (Bocconi, Chioccariello, Dettori, Ferrari, & Engelhardt, 2016) and engaging external vendors to conduct extra-curricular coding and computational thinking courses (Infocomm Media Development Authority, n.d.).

In the K-12 computational thinking curriculum domain, block-based environments are especially common instead of traditional text-based programming environments. Block-based programming environments capitalize on its visual nature, providing visual cues and only allowing blocks of code which form valid syntactic statements to be connected together (Weintrop & Wilensky, 2017). Block-based languages are less syntactic and sematic complex than traditional text-based languages (Simpkins, 2014) which allow young programmers to focus on more computational thinking aspects such as analysing the problem and devising a solution algorithm (Coravu, Marian, & Ganea, 2015).

Out of the many block-based programming languages, Scratch has been the popular choice for K-12 computational thinking education due to its simplicity for children with no prior experience with coding, its scalability where learners of varying skill level can create projects to highly complex applications (Dzhenzher, 2014).

2.4 Assessment of Computational Thinking Skills

In the International Computer and Information Literacy Study 2018 (ICILS 2018), Fraillon, Ainley, Schulz, Duckworth and Friedman (2018) summarized computational thinking into the following aspects which can be used for assessment:

1) Conceptualizing Problems
    a. Knowing about and understanding computer systems
    b. Formulating and analysing problems
    c. Collecting and representing relevant data
2) Operationalizing Solutions
    a. Planning and evaluating solutions
    b. Developing algorithms, programmes and interfaces

In an in-depth analysis of many sources, the definition is comparable to those used by other sources (Eickelmann, 2019). Where majority of the concept share its similarity in one form or another.

2.4.1 Model for Defining Computational Thinking

Lowe and Brophy (2017) identified distinct concepts and condensed them into 9 primary categories of

A) Abstraction, Decomposition, Pattern (recognition and generalization)

B) Algorithms, Data (collection, analysis and representation)

C) Parallelism, Simulation (and automation) and,

D) Testing and Debugging

They illustrated an operationalized concept map for computational thinking as shown in Figure 1 which shows how computational thinking elements are interlinked. They proposed that it presents potential for analysis projects which are unrelated to programming.

In the figure, a “human icon” is placed where there is possible interaction of the programmer to demonstrate their computational thinking abilities. By identifying the areas which the programmer participates in, analysis can be done to grade the programmers’ computational thinking ability.

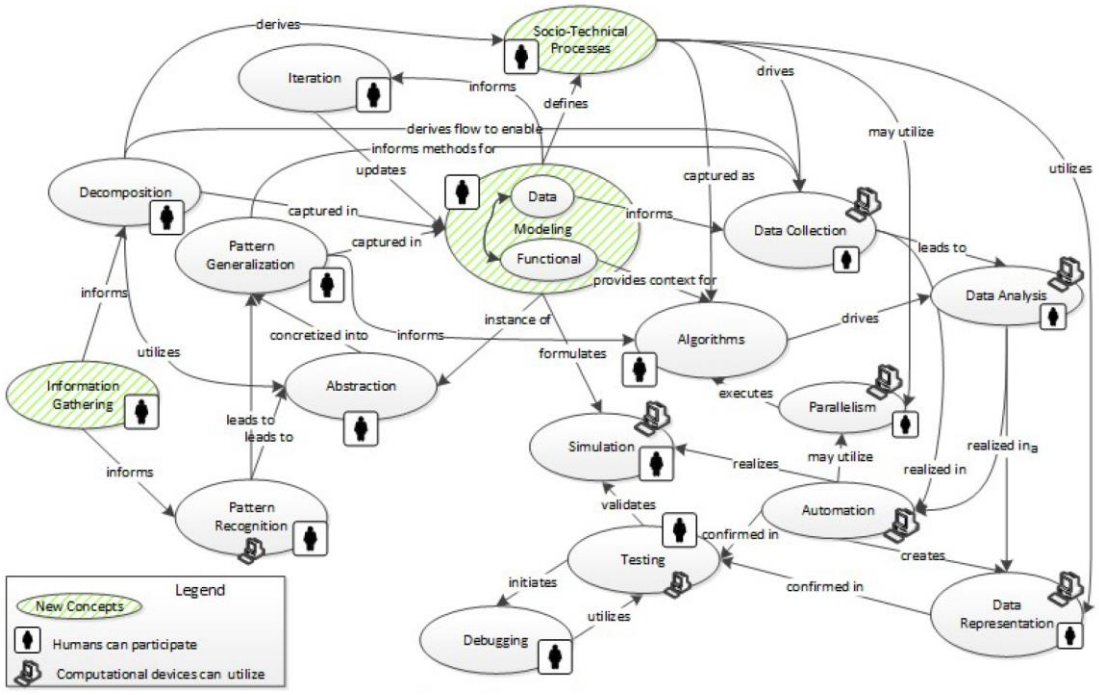


Figure 1 Operationalized concept map of CT (Lowe & Brophy, 2017)

2.4.2 Qualitative Analysis of Computational Thinking Concepts

Avila, Foss, Bordinim, Debacco and Cavalheiro (2019) proposed a rubric for evaluation of computational thinking with the main classification group of Abstraction, Decomposition, Generalization and Algorithmic Thinking. Within each group they designed criteria to grade the students computational thinking ability with scores from 0 to 5 as shown in Figure 2. The limitation of the rubric was identified using the case studies conducted in the study where the phrasing within each section was up to the interpretation of the assessors.

EVALUATION RUBRIC - CT ACTIVITIES

	0	1	2	3	4	5
ABS	No structures, activities or challenges were found that would lead students to structure processes of abstraction at any level, were found.	**Summary:** use of one abstraction layer. **Description:** the activity allows students to contact (or be presented to) representations of reality, abstractions in the form of drawings, maps, models or other formats and use this representation to solve some kind of problem.	**Summary:** use of more than one abstraction layer. **Description:** the activity allows students to contact (or be presented to) different levels of representations of the same reality, abstractions in the form of drawings, maps, models or other formats, and use these representations (available in more than one layer) to solve some kind of problem.	**Summary:** construction of one abstraction layer. **Description:** the activity allows students to engage in problem solving by constructing reality representation (abstraction), omitting details or detailing features in an abstraction layer.	**Summary:** construct more than one layer of abstraction. **Description:** the activity allows students to engage in problem solving by constructing more than one level of reality representation (abstractions), omitting detail or detailing features in more than one layer of abstraction.	**Summary:** definition of relations between layers of abstractions. **Description:** the activity requires that students engage in the definition/construction of relations between different levels of abstraction or in the construction of abstractions that satisfy certain relations.
DEC	It was not possible to identify that the didactic materials or activities promote or develop the competence of the decomposition.	**Summary:** use of composition/decomposition already established to solve problems. **Description:** the activity does not aim to work the decomposition, but students come into contact with (or are presented to) decomposing problems into subproblems or the composition of subproblems, and use this previously elaborated structure to solve some kind of problem.		**Summary:** use of previously defined subproblems, so that their composition can solve problems. **Description:** the activity allows the student to become involved in solving subproblems to solve a larger problem. The problem to be solved is already presented decomposed and the student must compose the subproblems to solve the problem as a whole.		**Summary:** identification of subproblems (decomposition) to solve problems (composition). **Description:** the activity requires the student to be involved in the division of the problem into smaller subproblems whose compound solutions lead to the solution of the larger problem. The decomposition and composition process is built by the student.
GEN	Activity does not involve the concept of generalization.	**Summary:** identification of patterns and similarities. **Description:** the activity aims to exercise the identification of patterns and similarities in problems, processes, solutions or data.		**Summary:** adaptation of solutions or part of solutions. **Description:** the activity aims to exercise the generalization in problem solving. Emphasis is placed on adapting solutions or parts of solutions, so that they apply to a whole class of similar problems.		**Summary:** creating solutions to solve a class of problems. **Description:** the emphasis of the activity is given in the creation of models or patterns that can solve a determined class of problems or in the creation of solutions that can be generalized.
AT	Activity does not involve algorithmic thinking.	**Summary:** use of instructions in sequence and manipulation of variables. **Description:** The activity aims to use instructions in a given order (sequences) and instructions that store, move and manipulate data to obtain the desired effect (variables and assignments).	**Summary:** use of control structures (selection / decision and repetition). **Description:** The activity targets the use of multiple-decision or repetition instructions, nested or not.	**Summary:** function building. **Description:** The objective of this activity is to create a collection of grouped and named instructions that perform a well-defined task (subroutines, procedures, functions, methods).	**Summary:** use of parallelism concepts. **Description:** The activity involves the identification of tasks that can be executed in parallel, considering the need for synchronizations, as well as the definition of the workload distribution between different execution flows.	**Summary:** creation of recursive algorithms. **Description**: The activity involves the definition and use of recursion as a form of iteration, considering recursive formulation of problems and/or application of the three laws of recursion (base case, base case approximation, recursive call).

Figure 2 Evaluation Rubric - CT Activities (Avila et al., 2019)

2.4.3 Analysis of Computational Thinking Ability with Process Data

Contrasting to the above methods, other works suggested that the programmer's actions during the course of their coding can be used as indicators of their computational thinking ability. Wang, Huang, Yan and Luo (2018) used the addition or deletion of blocks along with its timestamp in their analysis. Similarly, Kesselbacher and Bollin (2019) analysed the programmers input of the mouse and keyboard to classify students into their programming pattern and strategy (Trial & Error, Unfamiliarity, Late Abstraction and Subprogram) which indicates their computational thinking ability.

2.4.4 Analysis of Computational Thinking Ability with Static Code Analysis

Another common method for analysis was by doing static code analysis. Static analysis of code has been common place for text-based programming languages, but for block-based programming language it poses challenges as many of these block-based languages run within a graphic user interface environment (Chang, Sun, Wu, & Guizani, 2018). Various methods (described below) were designed to with its own method in solving this difficulty and allow analysis to be done.

Static code analysis can be carried out in many forms, for the web application Dr Scratch, the tool is built upon the python library Hairball with goals of allowing both the educator and the learner "to assess the quality of Scratch projects and detect

common errors and bad programming habits" (Moreno-León & Robles, 2015). By identifying the types of blocks which are present (Boe, et al., 2013), Dr Scratch grades the project on the computational thinking concepts that are used, with scores from 0 to 3. Thereafter it provides a summary these results as shown in Figure 3.

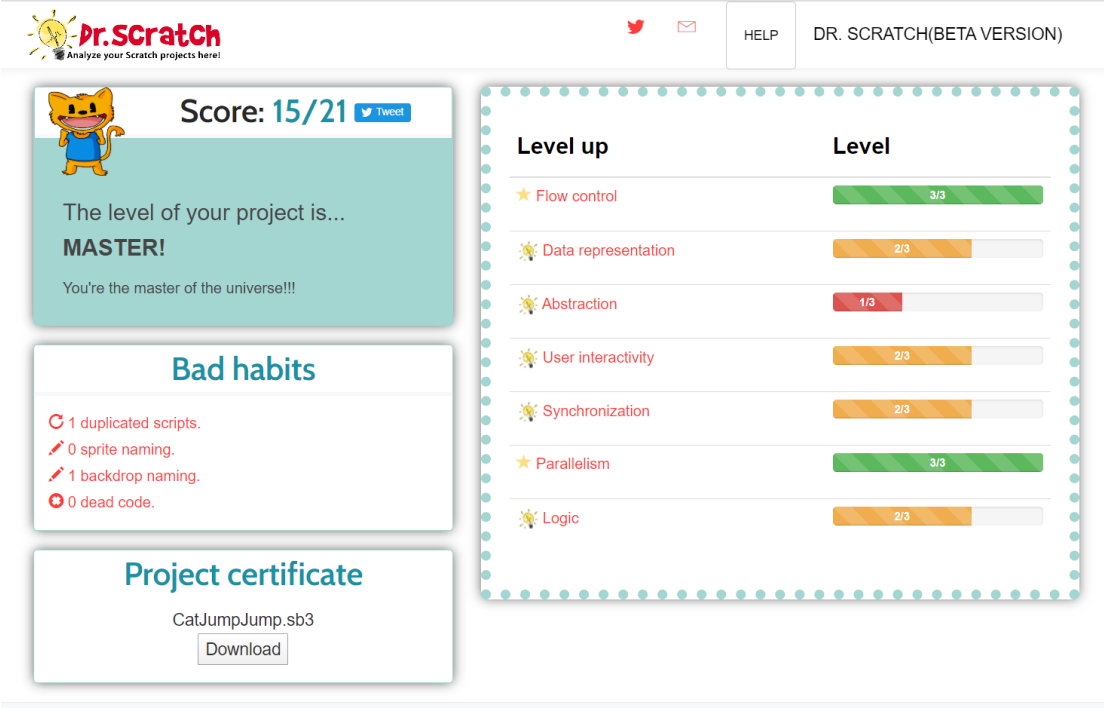


Figure 3 Example of Score Summary Page of Dr Scratch (Dr Scratch, n.d.)

2.4.5 Analysis of Computational Thinking Ability by Analysing Programme Behaviour

Instead of static analysis, programme behaviour can also indicate the programmers understanding of computational thinking concepts (Koh, Basawapatna, Nickerson, & Repenning, 2014). Programme behaviour on block-based programming environments can be implemented in the form of a debugger tool, such as in Whyline which implements Interrogative Debugging (Ko & Myers, 2004).

FINDINGS AND EVALUATION

The research conducted brought about many insights regarding the topic of computational thinking and the assessment of it.

Regarding the definition of computational thinking, most of the sources investigated share majority of the characteristics with the definition from Aho and Wing. Whereas existing works on assessment of computational thinking have criteria definition which coincides with the major characteristics and is comparable to the summarized form presented in ICILS 2018. Thus, this paper will adopt this definition as a guide.

From the Operationalized Concept Map for Computational Thinking, aspects in which the programmer is involved in are the areas which assessment of the programmer's computational thinking ability can be assessed.

By analysis of the categories that the programmer participates in:

A: Abstraction, Decomposition, Pattern (Recognition and Generalization)

B: Algorithms, Data (Collection, Analysis and Representation)

C: Parallelism, Simulation (and Automation)

D: Testing and Debugging

Qualitative analysis using a rubric enables assessment of all categories, but limitations include potential inaccuracies due to varying interpretation of the scoring criteria, requirement for the assessor to be well versed and highly trained in scoring, also it might be more difficult to automate the scoring as there are many combinations of in devising solution for a given problem.

Analysis using process data also enables assessment of all categories, but limitation includes inaccuracies due to many different user interaction of the programmer, also for the analysis to be accurate, programmers must be familiar in the programming environment unless it might indicate subobtimal computational thinking ability which might be more pronounced in the younger programmers of the K-12 range.

Analysis of static code enables assessment of A, B and C of the category but is limited for D. With static code analysis. Static code analysis might have its limitation as predicting programme behaviour might be hard, thus might not be indicative of how the programmer designed the solution.

Analysis of programme behaviour might be useful for A, B and C but is not effective for D if the analysis does not take into account how the programmer tests and debugs the solution.

The project will mainly focus on scratch since it is the most widely used open source block-based language. Scratch also runs on its own engine with structures that aid the user in developing solutions with abstraction in mind since the language has an object oriented design (Sprites, Stages etc). Since Scratch runs on many platforms as its own environment it allows accessibility to a wide user base. Despite selection of Scratch as the source block-based programming environment, this project will be designed to be easily extended to be compatible with other visual block-based languages

DESIGN AND IMPLEMENTATION

The assessment tool for this project is named as SB3 Analyzer. The term SB3 is chosen as it represents Scratch Blocks 3.0.

It can be retrieved from github.com/gnoeykeCG4001/sb3Analyzer

4.1 Overall Design

SB3 Analyzer is a collection of functionalities to enable easy analysis of Scratch projects. In SB3 Analyzer the data is stored in a format which static code analysis can be carried out easily. Even though the analysis and functionality which is currently operational works on static code, but also presents a potential for analysis of programme behaviour as it encompasses an engine which can step into the stages of the programme. SB3 Analyzer is designed to be able to dynamically update its data which allows expansion of the SB3 Analyzer do its analysis dynamically if real time information of the addition and deletion of blocks is available.

4.2 Architecture and Design Considerations

SB3 Analyzer has the following structure as shown in Figure 4. It is structured as a system of interconnected modules which is design in an object-oriented configuration. It allows modules to be expanded in the future and used separately supposing that their dependencies are also updated. The structure was designed with consideration of maximising compatibility with other block-based programming languages in the future.

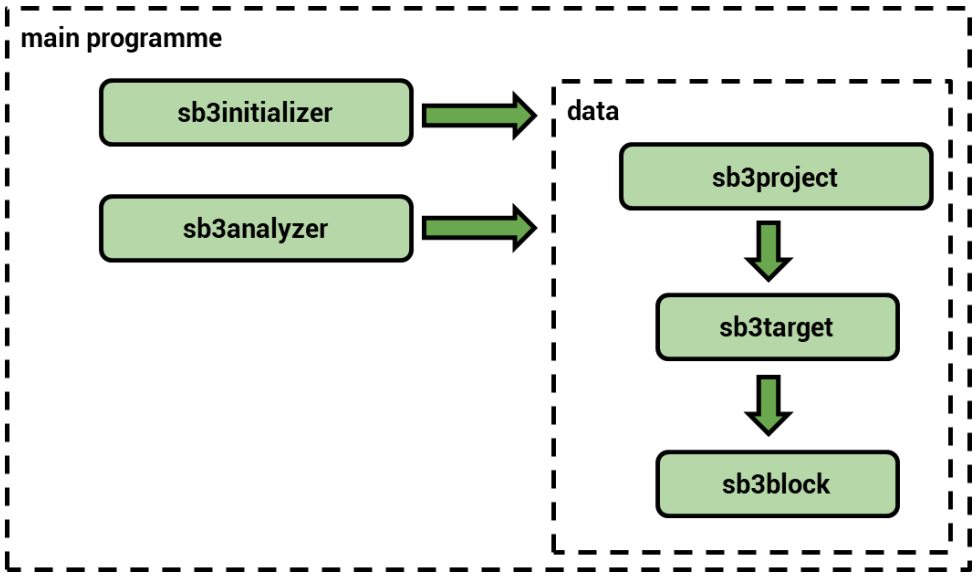


Figure 4 High Level Architecture of SB3 Analyzer

The main programme core comprises of programme configuration of SB3 Analyzer and provides the source location of the Scratch project. It then does basic initialization of the sb3initializer and sb3analyzer modules.

The sb3initializer module extracts the project data and store it in the data core format which prepares the data for analysis.

The sb3analyzer module contains the assessment schemes and the engine which allows iteration through the code which was previously generated. As it iterates through the code, it executes the selected assessments scheme on the project code.

4.3 Functionality of SB3 Analyzer

4.3.1 Conversion into Textual Representation

SB3 Analyzer allows the project to be converted into textual form as seen in Figure 5. This demonstrates the for block-based code to potential to be used with any external text-based programming language assessment tools. Configuration of SB3 Analyzer to output the textual representation in appropriate format for the external software is required for it to work.

Generated textual representation	Scratch Block in Scratch Environment

```
event_whenbroadcastreceived[jumpjump]()
control_if_else(operator_lt(noOfJumps, 3)){
        data_changevariableby[noOfJumps](1)
        control_repeat(2){
                motion_changeyby(20)
                control_wait(0.02)
        }
        control_repeat(2){
                motion_changeyby(10)
                control_wait(0.02)
        }
        control_wait(0.01)
        control_repeat(2){
                motion_changeyby(-10)
                control_wait(0.02)
        }
        control_repeat(2){
                motion_changeyby(-20)
                control_wait(0.02)
        }
        event_broadcast(land)
}{
        looks_sayforsecs(Im tired, 1)
        data_setvariableto[noOfJumps](0)
}
```

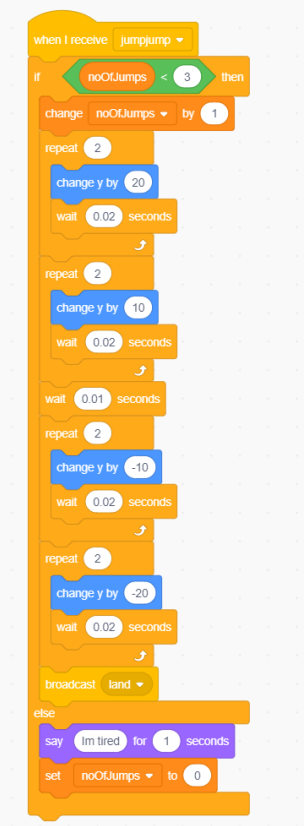


Figure 5 Comparison of Textual Representation vs Source Blocks

4.3.2 Static Analysis - Unreachable Blocks

SB3 Analyzer allows static analysis of the project, it can identify blocks which are unreachable or unused as seen in Figure 6.

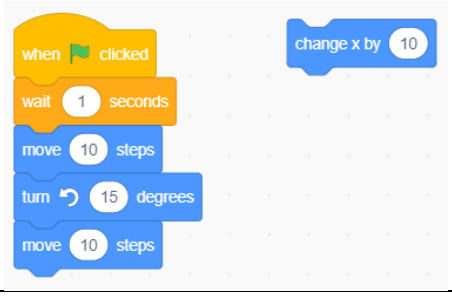
Identified unreachable code	Scratch Block in Scratch Environment
unreachable blocks: Z6h821*(?y;,)nI``/9m : motion_changexby	

Figure 6 Identification of Unreachable Code vs Source Blocks

4.3.2 Static Analysis – Matching Blocks

SB3 Analyzer allows static analysis of the project, it can identify blocks which matches certain criteria (e.g. if statements & if-else statements) as seen in Figure 7. This functionality allows analysis similar to Hairball and Dr Scratch which grades the project based on the usage of blocks which indicates a particular computational thinking ability (Moreno-León & Robles, 2015).

Identified matching code

```
matched blocks:
jss^qvF%FY[8G:]Po1Dy : control_if
V0z{5[KP#p)HRKp^Om^{ : control_if
0{lsu!8f+FN;du^)E0{q : control_if_else
```

Scratch Block in Scratch Environment
(The red dot indicates the blocks which matches the criteria)

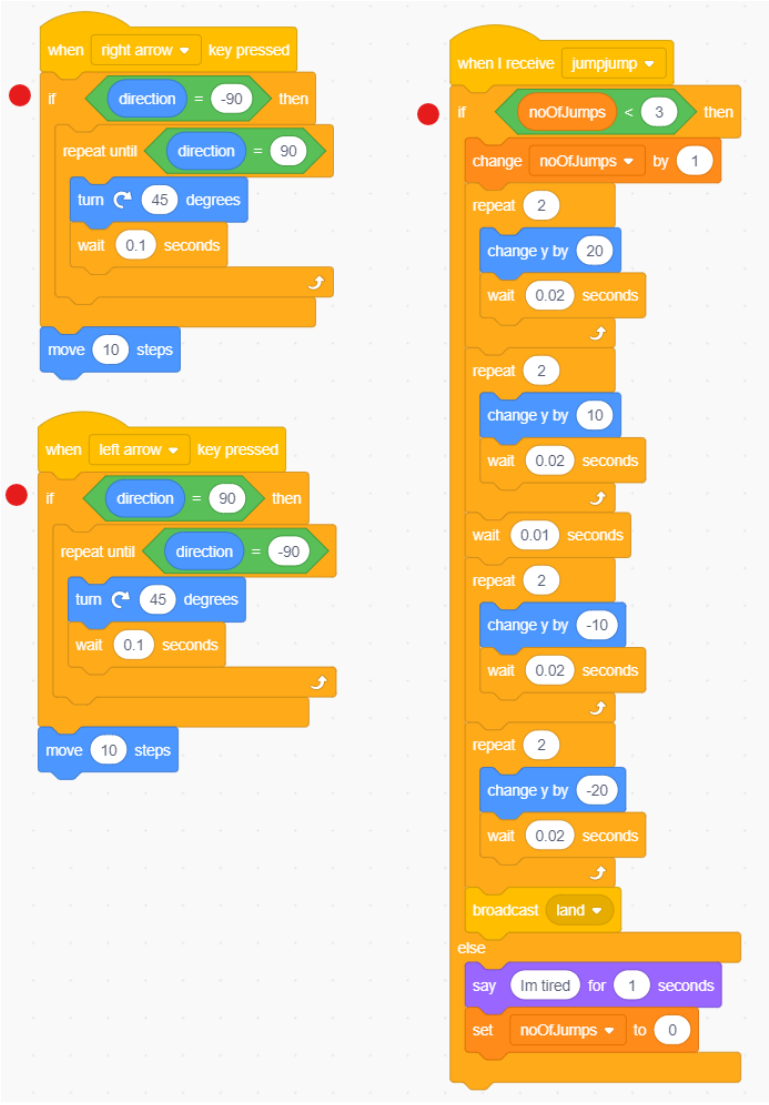


Figure 7 Identification of Matching Code vs Source Blocks

4.3.3 Other Features of SB3 Analyzer

As the SB3 Analyzer engine allows step wise iteration through the blocks, it simulates the actual programme execution stack allowing it to act as a debugging tool. This can be expanded to track variables throughout the execution process where the programmer can use it to test the operation of the programme.

CONCLUSION

6.1 Summary of Project

SB3 Analyzer is contains functionality to assist a wide range of assessment schemes. It currently is unable to give a score on a project in computational thinking aspects as a fixed rubric has not been integrated to the assessment scheme of the analyser module.

6.2 Future Works

1. Integrating more assessment schemes
2. Development of graphical user interface
3. Development of debugging tool
4. Development of an integrated development environment
5. Extending SB3 Analyzer to other block-based languages

REFERENCES

Aho, A. V. (2011). Computation and Computational Thinking. *The Computer Journal*.

Avila, C. O., Foss, L., Bordini, A., Debacco, M. S., & Cavalheiro, S. A. (2019). Evaluation Rubric for Computational Thinking Concepts. *IEEE 19th International Conference on Advanced Learning Technologies*.

Bocconi, S., Chioccariello, A., Dettori, G., Ferrari, A., & Engelhardt, K. (2016). *Developing Computational Thinking in Compulsory Education : Implications for policy and.*

Boe, B., Hill, C., Len, M., Dreschler, G., Conrad, P., & Franklin, D. (2013). Hairball: Lint-inspired static analysis of scratch projects.

Chang, Z., Sun, Y., Wu, T.-Y., & Guizani, M. (2018). Scratch Analysis Tool(SAT): A Modern Scratch Project Analysis Tool based on ANTLR to Assess Computational Thinking Skills.

Coravu, L., Marian, M., & Ganea, E. (2015). Scratch and recreational coding for kids.

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

Dr Scratch. (n.d.). *Dr Scratch*. Retrieved from http://www.drscratch.org/

Dzhenzher, V. O. (2014). Computer Simulation at School, Scratch and Programming Language Choosing Criteria. *2014 IEEE Global Engineering Education Conference (EDUCON)*.

Eickelmann, B. (2019). Measuring Secondary School Students' Competence in Computational Thinking in ICILS 2018—Challenges, Concepts, and Potential Implications for School Systems Around the World. In *Computational Thinking Education.* SpringerOpen.

Fraillon, J., Ainley, J., Schulz, W., Duckworth, D., & Friedman, T. (2018). Assessment Framework. *International Computer and Information Literacy Study 2018*.

Infocomm Media Development Authority. (n.d.). *Code for Fun Enrichment Programme aims to increase primary and secondary school students' exposure to computational thinking and making.* Retrieved from https://codesg.imda.gov.sg/in-schools/code-for-fun/overview/

K-12. (n.d.). Retrieved from Macmillan Dictionary: https://www.macmillandictionary.com/dictionary/british/k-12

Kesselbacher, M., & Bollin, A. (2019). Quantifying Patterns and Programming Strategies in Block-based Programming Environments. *IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings*.

Ko, A. J., & Myers, B. A. (2004). Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior.

Koh, K. H., Basawapatna, A., Nickerson, H., & Repenning, A. (2014). Real Time Assessment of Computational Thinking. *2014 IEEE Symposium on Visual Languages and Human-Centric Computing*.

Kong, S.-C., Abelson, H., & Ming, L. (2019). Introduction to Computational Thinking. In *Computational Thinking Education.* SpringerOpen.

Lowe, T., & Brophy, S. (2017). An Operationalized Model for Defining Computational Thinking.

Moreno-León, J., & Robles, G. (2015). Dr. Scratch: a Web Tool to Automatically Evaluate Scratch Projects.

National Research Council. (2012). *A Framework for K-12 Science Education: Practices, Crosscutting Concepts, and Core Ideas.* National Academies Press.

National Science Foundation. (2007). *NSF Announces $26 Million Solicitation for Projects That Advance Innovative Computational Thinking.* Retrieved from National Science Foundation: https://www.nsf.gov/news/news_summ.jsp?cntn_id=110134

Papert, S. (1971). *Teaching Children Thinking.* Massachusetts Institute of Technology A.I. Laboratory.

Purdue University. (n.d.). *Samuel D. Conte*. Retrieved from https://www.rcac.purdue.edu/compute/conte/bio/

Simpkins, N. (2014). I scratch and sense but can I program? An investigation of learning with a block based programming language. *International Journal of Information and Communication Technology Education*.

Tham, I. (2017). *Singapore to use micro:bit to teach coding, nurture its own Steve Jobs.* Retrieved from The Straits Times: https://www.straitstimes.com/singapore/singapore-to-use-microbit-to-teach-coding-nurture-its-own-steve-jobs

Wang, X., Huang, X., Yan, C., & Luo, H. (2018). Analysis of Scratch Project with Process Data.

Weintrop, D., & Wilensky, U. (2017). Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education*.

Williamson, B. (2016). Political computational thinking: policy networks, digital governance and 'learning to code'. *Critical Policy Studies*.

Wing, J. (2006). Computational Thinking. *Viewpoint, Communications of the ACM*.

Wing, J. (2010). Computational Thinking: What and Why?

Zink, B. (2002). Computer science pioneer Samuel D. Conte dies at 85. *Purdue News Service*.

APPENDIX A

Setting up SB3 Analyzer

The source code of the project retrieved from: github.com/gnoeykeCG4001/sb3Analyzer

1) Ensure that you have Python 3.8 installed

2) Extract the source code (from the source above)

3) In main.py assign the project file path into the *sb3filepath* variable

4) In sb3analyser.py uncomment the portion "***### X"***
   for the required functionality where X is:
   A) To generate a list of blocks in the project
   B) To generate textual representation of the project code
   C) To retrieve a list of unreachable blocks
   D) To retrieve matching blocks

5) Execute the programme by running *runMain.bat*

6) A copy of the output can be retrieved from *main_output.txt*