UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Ismat Alakbarov

Dynamic Analysis of Scratch Projects to Infer Computational Thinking Abilities

Master's Thesis (30 EAP)

Supervisor(s): Marcello Sarini

Co-Supervisor(s): Marlon Dumas

Tartu 2021

Dynamic Analysis of Scratch Projects to Infer Computational Thinking Abilities

Abstract:

The role of visual block-based programming languages has become prominent in children's computer science education in many schools across the globe, allowing children to concentrate on creating programs by eliminating syntactical program errors. Consequently, the necessity for auto-assessment systems has become apparent as the evaluating learner' projects required manual labour, which was placed on instructors' shoulders. Thus, numerous auto-assessment systems are built to assist instructors in evaluating students' computational thinking skills to cope with this increasing demand. Inspired by the existing literature review on this topic, we envision that behavioral similarity between Scratch programs and their code coverage could be used to infer the Computational Thinking skills of learners. Therefore, we built a web-based tool called DSEScratch that calculates three metrics of behavioral similarity and code coverage by employing dynamic symbolic execution. We anticipate that our system could complement existing Scratch analysis tools to gain deeper insights into learners' Computational Thinking skills.

Keywords:
Dynamic Analysis, Static Analysis, Dynamic Symbolic Execution, Scratch Analysis Tool, Behavioral Similarity

CERCS: P170: Computer science, numerical analysis, systems, control

Kraapimisprojektide dünaamiline analüüs arvutusliku mõtlemise võimete järeldamiseks

Lühikokkuvõte:

Visuaalsetel plokkidel põhinevate programmeerimiskeelte roll on laste informaatikaõppes esile kerkinud paljudes koolides üle kogu maailma, võimaldades lastel keskenduda programmide loomisele, kõrvaldades süntaktiliste programmide vead. Sellest tulenevalt on tekkinud vajadus automaatse hindamise süsteemide järele, kuna õppija projektide hindamine nõudis käsitsi tööd, mis pandi juhendajate õlgadele. Seega on ehitatud arvukalt automaatse hindamise süsteeme, mis aitavad juhendajatel õpilaste arvutusliku mõtlemise oskusi hinnata, et kasvava nõudlusega toime tulla. Selle teema kirjanduse ülevaatest näeme, et Scratch-programmide ja nende koodide katvuse sarnast käitumist võib kasutada õppijate arvutusliku mõtlemise oskuste hindamiseks. Seetõttu ehitasime veebipõhise tööriista nimega DSEScratch, mis arvutab dünaamilise sümboolse täitmise abil välja kolm käitumise sarnasuse ja koodide katvuse mõõdikut. Eeldame, et meie süsteem võiks täiendada olemasolevaid kriimustusanalüüsi tööriistu, et saada põhjalikumat ülevaadet õppijate arvutusliku mõtlemise oskustest.

Võtmesõnad:
Dünaamiline analüüs, staatiline analüüs, dünaamiline sümboolne täitmine, kriimustuste analüüsi tööriist, käitumissarnane sarnasus

CERCS: P170 - Arvutiteadus, arvanalüüs, süsteemid, juhti

Table of Contents

1 Introduction

As computing devices have been integrated into many aspects of our lives, the importance of acquiring the knowledge of computer science fundamentals at a young age has become apparent. Subsequently, several initiatives have been made into teaching computer science principles in childhood education from an early age, leading to computing-related courses being included in the school curriculum in many countries. Strictly related to learning programming skills, there is the theme of computational thinking, which is about solving problems algorithmically by using a programming language.

One of the initiators of teaching CT competencies is the K-12 computer science framework, a high-level set of guidelines that informs the development of standards, curriculum, course pathways, and professional development [1]. According to this framework, CT is not something unique to computer scientists and considered basic competence that everyone should possess. According to the framework, for the young generation to effectively develop CT skills, they should be taught computer science.

One of the most effective ways to teach computing is through creating and refining computer programs [2]–[4]. Thus, several block-based and text-based programming environments have been designed where children can get hands-on experience by formulating solutions as computational steps to be executed by computers. Due to common mistakes in coding in text-based programming languages, such as syntax errors, a visual block-based programming environment is generally the preferred choice of instructors and teaching assistants (TA). It provides suitable habitat for beginners/kids to explore the CT concepts by eliminating irrelevant details. Several visual block-based programming languages have been designed, such as Snap!, Scratch, App Inventor, Blocky.

Initially, submitted solutions of novice programmers to assigned programming tasks often manually checked by instructors or TAs, which was considerably time-consuming. Thus, there was a clear need for auto-assessment systems. Several approaches developed for automatically assessing projects created in block-based coding programs, many of which relied on code analysis to address this issue. The majority of the existing auto-assessment systems use static analysis, whereas only a few perform dynamic code analysis.

One of the primary use cases of dynamic code analysis is to check the correctness of a program and find out errors that occur during program execution. In this case, it is usually performed by executing the program subjected to predefined tests and comparing the outputted result with the

expected value. On the contrary, static code analyzers are concerned with the quality of source code by checking the code against a set of software metrics and making sure the solution follows the exercise instructions, for example, detecting the presence of a specific programming statement (e.g., for, while, if-else), etc. However, in the context of this thesis topic, we are interested in dynamic symbolic execution as an alternative to conventional dynamic and static analysis methods to get valuable measures. Traditional dynamic analysis is mainly used to check the functional correctness by executing it against a set of predefined test cases. In contrast, dynamic symbolic execution has several use-cases, such as achieving high code coverage, measuring behavioral similarities, uncovering edge cases not considered in the code implementation, etc.

In this thesis topic, the goal is to create a tool that calculates metrics by performing dynamic symbolic execution of code created in Scratch by novice programmers to infer their CT abilities that could potentially be used in assessing Scratch projects. The thesis consists of two parts. In the first part, a review of the literature on code analysis concerning the comparison of static and dynamic analysis and symbolic execution have been conducted to investigate meaningful measures that can be acquired during the execution of Scratch code. Furthermore, we also examined CT and its dimensions.

In the second part, based on the literature review on the analysis of both text-based and block-based programs, we described the design and implementation of our web tool[1], DSEScratch, that leverages dynamic symbolic execution (DSE) to compute meaningful metrics. To the best of our knowledge, leveraging DSE and measuring behavioral similarity in the context of Scratch analyzing or testing has not been done previously. Thus in our work, we proposed an innovative way of employing DSE to analyze code created with block-based visual programming languages.

[1] https://github.com/IsmatAl/thesisProject

2 Related work

2.1 Computational Thinking (CT)

This notion was first introduced by Seymour Papert in his book[5]. However, it gained popularity by J. Wing[6]. As the awareness of computational education rose it drew the attention of researchers all over the world and became the subject topic/focus of many research [7]–[9] which led to a number of definitions resulted in variations in its meaning. Hence "CT literature is at an early stage of maturity and is far from either explaining what CT is or how to teach and assess this skill" [7] and even referred to as a "blurry psychological construct"[10]. Because of this, its assessment becomes challenging [11]. As a result, several literature reviews have been conducted on studies of which CT was the subject topic.

Covering all the definition with its depth and breadth is out of the scope of this thesis topic. Nevertheless, the followings are some of the definitions that are noteworthy.

According to Aho, CT is "the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms" [12].

Inspired from an email exchange with Al Aho, J. Wing, again provided the following definition of CT[13]:

> *".. Computational Thinking is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent…"*

In their paper, Hoppe and Werneburg concluded the following[14] definition which is in line with Wing's point of view that CT shouldn't be understood as thinking like a computer but rather "it is a way humans solve a problem"[6]:

> *"… The essence of Computational Thinking (CT) lies in the creation of "logical artefacts" that externalize and reify human ideas in a form that can be interpreted and "run" on computers. Accordingly, CT sets a focus on computational abstractions and representations—i.e., the computational artefact and how it is constituted is of interest as such and not only as a model of some scientific phenomenon. …"*

On the other hand, an operational definition of CT developed by the International Society for Technology in Education (ISTE) and the Computer Science Teacher Association (CSTE) within the framework of K-12 Education includes the followings[15]:

- "Formulating problems in a way that enables us to use a computer and other tools to help solve them;"
- "Logically organizing and analyzing data;"
- "Representing data through abstractions, such as models and simulations;"
- "Automating solutions through algorithmic thinking (a series of ordered steps);"
- "Identifying, analyzing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources;"
- "Generalizing and transferring this problem-solving process to a wide variety of problems;"

The last bullet point suggests "students may be considered to be exhibiting computational thinking even though they are not creating with technology tools"[16]. Sysło and Kwiatkowska also highlighted that CT is a set of thinking skills that are not necessarily reflected only in computer programming. Furthermore, it should "focus on the principles of computing rather than on computer programming skills"[17].

In addition to the above definition, ISTE developed the vocabulary for CT: Data Collection, Data Analysis, Data Representation, Problem Decomposition, Abstraction, Algorithms & Procedures, Automation, Simulation, Parallelization[15].

2.1.1 CT dimensions

CT framework developed by Brennan and M. Resnick which has three dimensions[11]:

- Computational concepts
    - *"Sequence* – a sequence of instructions to be executed on a computational agent;"
    - *"Loops* – a mechanism for executing the same sequence several times;"
    - *"Parallelism* - sequences of instructions happening at the same time;"
    - *"Events* - one thing causing another thing to happen;"
    - *"Conditionals* - the ability to make decisions based on certain conditions;"
    - *"Operators* - mathematical, string, and logical expressions to perform numeric or string manipulations;"
    - *"Data* - storing, retrieving, and updating values;"

- Computational practices – concerned with thinking and learning processes, not only focusing on the things you are learning but also the practices you are developing in your learning experience
    - *"Being incremental and iterative* - an adaptive process, one in which the plan might change in response to approaching a solution in small steps;"
    - *"Testing and debugging* - it is critical for designers to develop strategies for dealing with – and anticipating – problems;"
    - *"Reusing and remixing* - support the development of critical code-reading capacities and provoke important questions about ownership and authorship;"
    - *"Abstracting and modularizing* - building something larger by putting together collections of smaller parts;"
- Computational perspectives
    - *"Expressing* – using computing as a medium that can be used to self-express and design;"
    - *"Connecting* – benefitting from access to others;"
    - *"Questioning* - interrogating the taken-for-granted, and, in some cases, responding to that interrogation through design;"

For National Research Council one's competences in cognitive, interpersonal and intrapersonal have to be assessed. The cognitive domain is represented by CT concept and practices, whereas the interpersonal and intrapersonal domains are represented by CT perspectives[18].

2.1.3 CT components and its evaluation methods

In his literature review of previous studies related to CT evaluation published from 2010 till 2019 May, Siu-Cheung Kong [19] attempted to identify CT concepts, practices and perspectives for evaluation in the CT framework proposed by Brennan and Resnick [11] and appropriate methods for their evaluation. In addition, he also proposed two evaluation components: 1) Problem formulation as a component of CT concepts; 2) Computational identity and programming empowerment as components of CT perspectives. S. Kong argued that formulating the problem is the indication of creativity and included it as a component of CT concepts. As far as programming empowerment is concerned, he defined it as the experiences of a person creating and developing programs that allow them to address real-life problems and participate confidently in the digital

world. He also defined computational identity as having a positive perception of competencies in attitudes, knowledge, character and skills in the computational context. His study' results are listed in Table 1.

№	CT dimensions	Components [2]	Evaluation methods	
			Quantitative	Qualitative
1	CT concepts	• Procedures • Initialization • Problem formulation	• multiple choice type questions • Task/project rubrics	• Interviews • Project analysis • Observations • Reflection reports
2	CT practices	• algorithmic thinking • problem decomposition • planning and designing	• Task/project rubrics • Tests designed with task-based questions	• Interviews • Observations • Reflection reports
3	CT perspectives[3]	• Computational identity • Programming empowerment	• Survey	• Interview

Table 1: CT dimensions, its components, and proposed methods for their assessment.

Qualitative and quantitative methods assess the components, as mentioned earlier, of all the CT dimensions. One way quantitative data could be acquired is by using rubrics. Rubrics use measurable attributes to set apart performance levels in achieving learning outcomes by defining the different criteria associated with learning activities and indicators distinguishing each level to assess the students' performance [20], [21]. For example, in the evaluation of CT practice components, the criterion is CT practice components. However, designing descriptors of performance levels for each CT practice is needed. Learners' competencies in formulating and solving problems (CT practice) could be reviewed using these rubrics. Moreover, multiple-choice test questions can be used as a quantitative method for assessing a large number of learners' understanding of CT concepts [19].

[2] In addition to the evaluation components specified in [11] The column contains others that mentioned as evaluation components of their respective CT dimension in other papers reviewed by Siu-Cheung Kong

[3] The evaluation components of CT perspectives are proposed by Siu-Cheung Kong

As far as qualitative methods are concerned, interviews with learners can be conducted. Learners are asked to explain their code to uncover conceptual gaps in their understanding of CT concepts [22]. Interviews can also be helpful to learn young programmers' "thoughts behind the code" (CT practice) and their attitudes towards programming (CT perspective) [19]. Learners can also write reflective reports better to comprehend their understanding of CT concepts and practices. However, it requires significant time and can be used as a supplementary tool to quantitative methods [23].

Furthermore, block usage information, namely, "the number, the range, and the frequency," can be used to assess novice programmers' development in CT concepts which can be obtained using tools that perform code analysis [24].

Another possible modality of evaluation of CT dimensions relies on the use of some forms of code analysis, as described in the following.

2.2 Code analysis in assessment systems

As the need for auto-grading systems rapidly increases, numerous methods to evaluate and assess the submitted code emerged. The majority of the existing techniques used by these systems employ code analysis which could be categorized into two main types: static and dynamic code analysis. Hybrid approaches that combine the two also exist.

2.2.1 Dynamic assessment

A dynamic analysis observes an execution of a program and analyzes the observations made during the execution. In most cases where this technique is used, the submitted code is tested against a set of predefined test cases. Then, results are compared with the expected output, which is the most preferred way of testing functional correctness.

Several universities primarily employ this technique for auto-grading purposes of their programming classes. Besides this, it is used in various auto assessment tools built for visual block-based languages as well. Example of this includes ITCH[27], Quizly[28], Wisker tool[26].

It is often the case that while dynamic analysis is performed, the results generated from the analyzed code is checked for its equality with the expected output. Additionally, the inability to provide an assessment in case of a program that does not compile or complete its execution (e.g. infinite loop) or generate output is one of the significant drawbacks of this approach.

2.2.2 Static Assessment

Dynamic techniques cannot analyze how the source code is structured and organized. In contrast, static analysis methods use technology built for language-based tools and compilers, analyzing structural quality issues and gathering details of the code without executing program code [29].

Furthermore, it enables identifying any violation against the rules defined in exercise instructions the code under test might contain. For example, consider a case that code contains only a single line that prints or returns the answer or contains built-in methods or data types not allowed according to the exercise instructions. Auto grading systems using solely dynamic approaches are incapable of detecting these violations and will consider such cases as successful even if the assignment requirements are not satisfied.

The static analysis often highlights typical errors, which are built-in most tools and are usable for teaching novice programmers, thus easing their learning curve. Its application varies from assessment of programming style, plagiarism or keyword detection, software metrics, detection of syntax and semantic errors, structural or non-structural similarity analysis [30][31][32].

Static approaches on their own are incapable of measuring some key CT competencies, such as remixing, designing or debugging skills. Thus, it led to a growing number of emerging hybrid solutions combining both approaches.

2.2.3 Hybrid Assessment.

Generally, static approaches solely analyze the quality of the source code, and the correctness of program code is left aside. On the contrary, conventional dynamic systems merely check the functionality of the code under assessment. The limitations mentioned above of both techniques led to constructions of hybrid systems that combined both approaches, thus added extra value relevant for code evaluation. Examples of such solutions include eGrader[33] for assessment of introductory courses in Java, Quimera[34] and AutoLEP[35] for source code written in C, and BOSS[36] for Java programs. ITCH[27] is an example of such hybrid solutions for Scratch programs.

2.2.4 Symbolic execution

Classical symbolic execution (SE), used extensively for program analysis purposes can be shown as an example of static code analysis technique[37]. It was first developed in the mid-1970s to see whether a piece of software might violate specific properties [38]–[41]. It gained practical value, especially in program analysis studies and several industry tools with recent advancements in hardware and SAT/SMT solvers. In particular, symbolic execution techniques may fall into two general categories:

Purely Symbolic Execution. Purely symbol execution, aims to assign symbolic values to variables solely for simulation reasons rather than actually running the program. While symbolic execution is performed on a program under test, the program's different execution paths are explored one at a time by the symbolic execution (SE) engine. Then a first-order Boolean formula is generated for each explored path, also known as path condition, that defines a set of inputs. When the program is passed any input value from the set defined by the formula, it will follow the same execution path. In case the generated formula for a particular execution path is unsatisfiable, the input set is empty.

On the other hand, if the path condition is satisfiable, then the input set it defines is not empty, and the path is feasible[42]. Thus, symbolic execution can be seen as a method of abstractly executing a program. One such execution covers several possible program inputs, defined symbolically in a path condition, that share a common execution path.

The result is then translated to an appropriate input format to feed to Automated Theorem Prover (ATP), such as Z3[43] that determines, for example, if any concrete values can be assigned to symbolically defined inputs, and yields a response of "satisfiable", "unsatisfiable", or "unknown".

A significant drawback of classical SE is its inability to produce input values for a path condition that cannot be solved by ATP[44]. In order to overcome this drawback and use symbolic execution in practice, a relatively modern SE technique - a combination of concrete and symbolic execution, can be used. Hence another term for it is concolic execution.

Dynamic Symbolic Execution. Dynamic Symbolic Execution (DSE), a relatively new technique built on top of purely symbolic execution was introduced as an example of hybrid code analysis technique, blending static and dynamic analysis to achieve the best outcome. It has several use cases such as white-box test generation, achieving high code coverage, as it tries to execute all

reachable statements in the program; for verifying program correctness. The sole difference between purely SE and DSE is that when the SMT solver cannot generate values that satisfy a complex condition, variables represented symbolically are substituted with concrete values, simplifying the path conditions.

DSE, also known as concolic (concrete and symbolic) testing, is driven by inputs that are set as default or chosen arbitrarily [42], [44], [45]. In other words, in concolic testing, SE is performed along the path taken by the program under analysis based on concrete inputs. As branches are explored, a new path constraint is extracted from its condition and added into a set of path constraints gathered previously along with the current execution. The conjunction of these constraints forms a path condition that an SMT solver analyzes to determine if the path condition is feasible. This is done by checking if any values exist that, when given as inputs to the path condition, outputs true. The most recent part of the path condition is then negated and again given to the SMT solver as input, and the process is repeated. The newly generated input is then given to the program under test, and a new path condition is generated. These steps continue until a preset code coverage is achieved or all feasible paths are explored[44].

```
1.  def  (x, y):
2.      if (x > y):
3.          return 1
4.      elif (x < y):
5.          return -1
6.      return 0
```

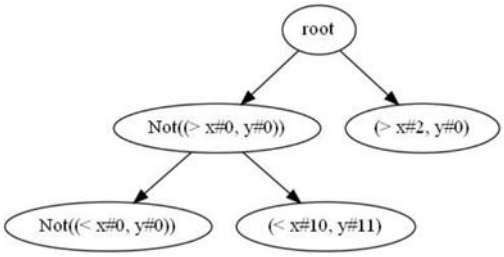


Figure 1: Easy example comparing two numbers and its execution flow

If concolic testing is applied on compare method in Figure 1, the execution will start with random input, say {0, 0}. When concrete execution reaches the if statement on line 2, a symbolic constraint ($x \leq y$) is generated that forms the path condition. Part of the path condition is then negated ($x > y$) and translated to a language format ATP can interpret. ATP is then invoked, and input values {2, 0} satisfying the path condition is generated. The program code takes a new path based on the newly generated input. After the path is explored, (not (x > y)) ∧ (not (x < y)) and (not (x > y)) ∧ (x < y) path conditions are generated and solved by ATP respectively. As a result, New input {10, 11} is produced and given to the source program, and finally, all paths are

explored, after which input generation is stopped. During concolic testing, path exploration is performed by following a depth-first strategy[44].

DSE is used to auto-generate inputs for calculating meaningful metrics for the thesis project which is discussed in detail in the later sections.

2.3 Code analysis methods in practice

2.3.1 Code Assessment Systems for visual block-based programs

Numerous studies on similar topics have been carried out that dealt with various aspects and forms of performance assessment of programming activities to infer CT abilities. However, only a tiny proportion of the studies focus on programs created in visual block-based programming languages that involve static and dynamic code analysis. Some of these studies have been listed in Table 2.

ID	Article Title
1	ITCH Individual testing of computer homework for scratch assignments
2	Quizly – A live coding assessment platform for App Inventor
3	Automatic detection of bad programming habits in scratch A preliminary study
4	Dr. Scratch – A web tool to automatically evaluate scratch projects
5	Comparing Computational Thinking Development Assessment Scores with Software Complexity Metrics
6	On the Automatic Assessment of Computational Thinking Skills – A Comparison with Human Experts
7	Assessment of Computer Science Learning in a Scratch-Based Outreach Program

8	Hairball Lint-inspired Static Analysis of Scratch Projects
9	Scrape: A tool for visualizing the code of scratch programs

Table 2: Related articles that involve static and/or dynamic code analysis.

The majority of the approaches in Table 2 are designed for assessing Scratch projects and involve static code analysis. The approaches differ according to the type of programming activity (Table 3).

Id	**name of approach**	**block-based visual programming language**	**type of programming activity**		**Automated analysis**	
			Open-ended well-structured problem with a pre-known correct solution	Open-ended ill-structured problem without a pre-known correct solution	Static	Dynamic
1	ITCH	Scratch	X		X	X
2	Quizly	App Inventor	X			X
9	Scrape	Scratch		X	X	
7 AND 8	Hairball	Scratch	X		X	
3, 4, 5, 6	Dr. Scratch	Scratch		X	X	

Table 3: Characteristics of the approaches employing code analysis.

It is possible to compare the program code of the students with representations of correct implementations for the given problem only in case of open-ended well-structured problems with a solution known in advance, which allows for dynamic code analyses[46].

Approaches concerned with open-ended ill-structured problems make use of static analysis, such as detecting the presence of a code block, which enables to identify which code blocks are used and calculate the frequency of their usage in the program. In order to infer CT skills, static analyzers search for the presence of program commands or constructs in program code to identify the learning of algorithms and programming concepts. This approach is based on the assumption

that the presence of a code element indicates an understanding of a concept, which is not necessarily true[47].

According to the K12 computer science framework, understanding the core concepts of computer science is considered computationally literate[1]. This view is accepted by most of the approaches presented in table 4, which attempt to identify the sub-concepts of one of the core concepts - algorithm and programming by searching for a presence of a specific programming statement or an algorithm. These sub-concepts are Control, Algorithm, Variables, and Modularity. The followings are brief information about the four sub-concepts [1]:

Control. The importance of execution flow of program instructions is very high, and it is specified by control statements, such as loops, conditional statements, error handlers;

Algorithm. An algorithm is a set of steps to solve a problem or achieve a specific result. A good algorithm is easy to implement, reusable, and has a high performance. There are specific algorithms that can be applied to many situations, and knowledge of these algorithms can enhance the ability to produce much more quality code;

Variables: Variables are used to store data, and the chosen data structure or variable type determines what kind of actions can be performed on a variable. Therefore, a good understanding of data structures reflects on the quality of the code;

Modularity. Modularity in code means readable, reusable code. Procedures can be shown as an example of modularity in software development;

Each of these sub-concepts reflects different aspects of code.

Approach	**Analyzed elements**					
	In relation to the CSTA K-12 curriculum framework				Additional elements	
	Algorithms	Variables	Control	Modularity	Functionality	Initialization
Hairball	X		X			X
Dr.Scratch	X	X	X	X		
ITCH	X	X	X	X	X	

Scrape	X	X	X	X		X
Quizly	X	X	X	X	X	

Table 4: Analyzed elements

Scrape[48] is a scratch-focused analysis tool that can perform analysis in a single Scratch project or across multiple projects and presents information about block usage, such as the number of times each code element used per category in a project code.

Hairball[49] is a lint-like tool that performs static analysis of Scratch projects. It was implemented as a set of Python scripts designed around an object-oriented paradigm to ease the adaption for evaluating specific tasks. It identifies code smells (e.g. duplicated codes, long scripts), potential errors, or unsafe practices, such as not initializing a variable or a character state, code that is never executed, and checking for the complexity of programs.

Dr. Scratch[50] is an open-source, web-based static analysis tool inspired by Scrape and based on Hairball. It allows performing static analysis of Scratch projects and is concerned with flawed programming practices and more abstract concepts such as parallelism, synchronization, data representation, user interactivity, and flow control. Based on the analysis, it assigns a score between 0 and 3 on these abstract concepts.

In one way or other, the approaches mentioned above are based on perceptive analysis of the kind of blocks used, identification of bad programming practice, potential issues, understanding of CT concepts by performing statistical analysis of code element usage. However, using only static analysis might have some disadvantages; for example, the functional correctness is left out.

The idea of testing Scratch programs was addressed in the form of the ITCH (Individual Testing of Software Homework for Scratch Assignments)[27] method that converts Scratch programs to Python code and then performs testing on Python code. It is implemented as a set of Python scripts and combines dynamic and static analysis. When performing static analysis, the ITCH can check and report on the presence of certain blocks, such as the custom block call. Routines are also available for counting loops, nesting loops, and checking sprite visibility.

The ITCH system provides test inputs and stores the test results with the help of block replacement. It is accomplished by using a property: when a Scratch file is saved, all variable values and the state of the Scratch world are stored as part of the file. Generally, blocks that provide temporary output to the screen are replaced by blocks that save output and blocks that need input

replaced by testing values. In other words, it is limited to testing inputs and outputs for "ask" and "say" blocks, thus supporting only a small subset of Scratch functionalities. Furthermore, no automated evaluation is provided by the ITCH system, just reporting results.

Quizly [28] is a part of an assessment system that allows for the formative, summative and informal evaluation of computer science skills through a visual language called App Inventor. One of the platform's features is user activity logging, which enables educators to monitor the progress of the student's learning path and their solution-building approach. User activity is logged with a time indication by the logging mechanism of the platform. Logged user activities can be ranged from inserting a block in the workspace, cancelling a block from the workspace, connecting or disconnecting two blocks and so on. This statistical information is later used to objectively assess the solution building process of users and eventually to identify user difficulties. Finally, the submitted solution is compared with the correct one prepared by the teacher to inform the student whether or not his/her submission is correct.

Whisker tool [26] provides automated and property-based testing functionality and supports manually written tests for Scratch programs. An expected and correct behavior is described as a set of observed properties for any violation when testing performed. Test subjects are stimulated using inputs generated randomly by a dynamic test harness and observed for conformance to the corresponding properties. As an input, Whisker accepts a specification (a set of properties) and one or more Scratch programs and produces two sets: a set of violated properties, each coupled with an error witness, and conditionally satisfying properties. Additionally, it can also generate a coverage report and a report in the TAP13 format to summarize the performed tests.

Whisker already achieves 95.25 per cent code coverage on average with its random testing approach, based on the findings of empirical evaluations on students and teachers. However, test flakiness - a case in which a test case produces different results when executed repeatedly, in the Scratch testing scenario when the random number generator is not seeded.

One of the cases the Whisker tool could be used is to support learners with fault localization. Fault localization is the act of identifying program locations that lead to erroneous state transitions, which in turn cause observable program failures[51]. As it is time-consuming activity[52], [53], which requires expertise[54], [55] for the programmers, a number of automated fault localization methods[3], [51], [56]–[63] have been created. All of these approaches are devised for programs written in text-based programming languages (e.g. Java, C).

According to the study, there is a moderate correlation between the grade points assigned by instructors to students' submissions and the number of test cases auto-generated by the Whisker tool the submissions pass, which suggests that better solutions pass a higher number of test cases thus achieve higher code coverage. Based on this finding, and assuming that in the grading process of the submissions, students' algorithmic thinking, planning and designing abilities were taken into account which corresponds to the CT practice dimension of the CT framework proposed by Brennan and Resnick we decided to add a feature to measure code coverage of Scratch programs to our tool.

Among the existing auto assessment systems examined above, only a few perform dynamic analysis of Scratch codes, such as the Whisker tool and ITCH. Unlike the ITCH tool, which runs in CLI, our system runs on a browser and offers additional functionalities, such as preparing code coverage reports. Furthermore, DSEScratch can be used to measure if Scratch projects are behaviorally similar. In addition, ITCH requires a user-supplied set of inputs for testing to check functional correctness, which requires manual labour. However, with our system, the need for human labour is significantly reduced as testing is automated using dynamic symbolic execution, enabling us to analyse many more projects than instructors previously did manually. Furthermore, unlike the Whisker tool, our system does not require a set of properties to be tested. However, as input, it requires at least two Scratch programs to calculate BS metrics and only one Scratch project to generate a code coverage report. Nonetheless, we think our tool could be combined with static analysis tools, such as the Dr Scratch tool, to get a clearer image of a student's CT skills.

2.3.2 Application of symbolic execution in code analysis

Pex4Fun[64], PyExZ3[42], BESIM[25], Code Hunt[65] are among many tools that utilize DSE in code analysis for educational purposes.

Pex for fun (Pex4Fun) was created as a web-based platform available from any device that can access the internet to learn and practice computer science skills such as abstraction and problem-solving through gaming. On this platform, users were able to exercise fundamental programming skills such as loops, arrays, exception handling by playing games devised by the tool's developers.

The underlying technology used for the Pex4Fun is a tool called Pex for auto-generating white box tests for .Net that employs DSE. The Pex tool uses a concept called parameterized unit testing, which has two main advantages. One, it specifies the methods' behaviors that are tested for a given

range of argument values. Two, a set of unit tests can be constructed by feeding the methods of the parameterized unit tests with argument values provided by Pex.

Pex4Fun allows its users to edit a method definition named Puzzle, which is analyzed using DSE by clicking on the "Ask Pex!" button just under the code editor. As a result, a table consisting of DSE generated input-output pairs is shown to the user. This set of input-output pairs can uncover interesting corner cases that the user does not consider. By practising on this platform, users had to study the code and generate test values manually, which help them acquire an understanding of a program and gain the ability to simulate programs in their minds.

A code duel was introduced, in which a user's method implementation is compared to a hidden answer. The method under test is given a set of input values generated by DSE on the secret implementation, and the outputs of the two are compared. It allows users to practice abstraction and program comprehension, problem-solving, and programming skills.

Code Hunt[65] is developed by Microsoft and based on the same concepts used in Pex4Fun.

PyExZ3[42] is a redesign of the NICE project's SE engine for codes written in Python, which now uses the Z3 theorem prover. In addition, all Nice-specific dependencies, codes specific to platforms are eliminated, and several other enhancements are made. A detailed overview of the tool's architecture has been provided in the paper "*Deconstructing Dynamic Symbolic Execution"*. The tool tries to explore all the paths in a Python function that are feasible by running it with a set of concrete input values and tracing the path that the function takes through its control flow. Then SE is performed on the traced path to generate new input values to explore untraced paths the program under test can take using Z3.

PyExZ3 might be able to explore all the paths the program can take for small methods that do not contain recursions or loops.

In their paper[25], Sihan Li, Xusheng Xiao and others proposed three metrics – Random Sampling (RS), Single-program Symbolic Execution (SSE), and Paired-program Symbolic Execution (PSE) to measure Behavioral Similarity (BS) between programs written in C#. According to the authors, BS can be used for various purposes, such as assessing students' code. A higher BS means a higher score. It can also be used as a progress indicator for a student. An increase of the indicator implies that students are on the right track, while a significant decrease means students' implementations behaviorally deviate from the solution.

A naive way to measure BS would be to go through the input domain and execute all input values by giving them programs under analysis and comparing their results. For obvious reasons, this would not be a feasible method depending on a program; the input domain might be infinite (e.g., it contains an infinite loop) or very large. Two approaches are adopted to solve this issue. First, taking advantage of the limited number of paths the reference program can take is much smaller than the input domain. A set of inputs that share common characteristics can cause the execution flow to run through a particular path. Instead of feeding the whole input set that results in a program taking the same path, specific inputs that exercise different execution paths can be used to calculate BS. These representative inputs can be acquired using DSE. Second, by randomly sampling input values uniformly from the input space. These insights allow users to calculate not the exact BS but to approximate it.

Random Sampling. This metric is calculated by using the second approach mentioned above. The advantage of this approach is that it is less costly compared to the other two metrics as the programs are not analyzed and treated as a black box. This is proved to be cost-effective and provides a good approximation in the calculation of BS. However, unlike SSE and PSE, when this approach is used, interesting inputs leading to deviations between programs under analysis might be missed due to input values being generated randomly.

Single-program Symbolic Execution. SSE is calculated using the first approach mentioned above. First, inputs and their corresponding outputs are generated by applying DSE on the reference program and are given to the program under analysis to compare the produced outputs. Then, SSE is calculated by computing the proportion of inputs resulting in the same outputs over all the DSE generated inputs. Thanks to DSE, the edge cases missed by the RS can be uncovered by SSE easily.

One of the drawbacks of SSE is that it considers two programs behaviorally similar or equivalent when all the DSE generated inputs leads to the same outputs. It does not necessarily mean that all the behaviors the program under analysis possess are covered, as only the behaviors of the reference program are captured.

Paired-program Symbolic Execution. Before PSE is calculated, the programs are paired, as shown in figure 2.

```
1. public void PairedProgram (object[] args) }
2.     Debug.Assert(Program1(args) == Program2(args));
3. }
```

Figure 2 Paired program example[25]

DSE is applied to the paired program, and inputs are generated. As the input domain of the newly formed program encompasses both the reference program and the program under analysis, it captures the behaviors of both of them. The results of the two are checked if they are equal. The number of inputs asserted to be equal over all the auto-generated inputs is PSE. Compared to the other two metrics, calculating PSE is relatively costly.

RS, SSE, and PSE metrics are initially intended to be measured for a text-based programming language (C#). However, as part of this thesis topic, we decided to adopt these metrics and code coverage and add them to the project. We anticipate this metric could be used as a factor in the assessment of Scratch projects. It might indicate how well a student understands a particular algorithm necessary to do his/her assignment. We did not find a similar tool that applies these techniques to code created with a visual programming language to our best knowledge during the literature review. Thus, with this tool, we provide an innovative way of Scratch code analysis.

3 Application Architecture

3.1 System Design

We envision that DSEScratch could be used as a complementary tool to existing CT evaluation methods described in Table 1. For example, reflection reports written by students on their submissions could help an instructor see whether they understand a task and express verbally an algorithm to solve the task. Combined with our tool, instructors can see if students lack programming skills to implement the algorithm described in their reflection. Furthermore, our tool could get clues about students' understanding of some of the computational concepts (e.g., sequence, loops, conditionals, operators, data) proposed by Brennan and M. Resnick. For instance, let us assume that students are assigned a task, which they should use loops and conditional statements to achieve the desired outcome. With static analysis, we can detect the presence of a for loop or if-then-else blocks. However, based on static analysis results, we cannot determine whether they are correctly used. We anticipate BS metrics, especially SSE, which indicates the submitted implementation behaves similarly to a reference program, are very high, we can reasonably assume that these statements are correctly used. In addition, based on the findings of [26], there is a high correlation between the grades and code coverage as a metric; in other words, higher code coverage generally signifies a higher grade.

With dynamic analysis of Scratch programs, to our best knowledge, it is impossible to gather block usage information but with the static analysis, it is possible. We assume the only information we can get with dynamic analysis is the one that is generated during the execution of Scratch projects. Based on the literature review, we concluded that dynamic analysis is usually used for functional correctness, such as the ITCH tool. During the literature review, we encountered DSE that seemed very promising to be used in the context of this thesis topic. We think it is indeed a new way of analyzing Scratch projects.

In addition, students could use our tool to uncover edge cases that they have not considered, motivating them to analyse and detect the bug and thus, exercise their debugging skills.

The project was built for as a web tool to compute metrics mentioned in the previous section and could be used by instructors in the assessment process of simple Scratch projects. It can be used for open-ended well-structured problem with a correct solution provided by an instructor. For a more comprehensive evaluation, the tool can be combined with well-known static analysis tools,

such as Dr Scratch, as it is impossible to check functional correctness or measure behavioral similarity and code coverage with the latter. It comprises of backend and frontend parts using Django framework in the backend, and React stack in the frontend side of the tool.

Due to Scratch programs are executed in a graphical user interface acquiring necessary data to compute aforementioned metrics is challenging. However, a Scratch project can be downloaded in a ZIP folder containing a JSON file storing all the block-usage information and other resource files, a property exploited to overcome this issue. During the process of designing the system, it was decided to use this property to convert a Scratch project to a text-based code as several tools perform DSE of code written in popular text-based programming languages, such as JavaScript (ExpoSE[66], SymJS[67]), Python (PyXZ3[42]), .Net (Pex[68]). After examining these systems, we found out that using PyExZ3 is more suitable for our case than the rest due to its simplicity. The others appeared to be relatively tedious to install and use. Furthermore, Python was chosen as our tool target language due to its flexibility, ease of use, and wide range of applications.

In order to convert Scratch to its Python equivalent, we decided to use ANTLR [69] (**AN**other **T**ool for **L**anguage **R**ecognition), a powerful parser generator, to generate a parser using customized ANTLR JSON grammar to parse extracted JSON files. In addition, Jinja2[4], a template engine to input parsed values into template string literals, was used to generate Python code.

3.2 System architecture

Figure 3 illustrates the system module for the analysis procedure. First, a Scratch project created by dragging and dropping is extracted, and the "project.json" file is uploaded on a web page to the system as input. Then the system processes the input files and calculates the requested BS metrics. The results are then displayed with graphs with generated data on a web page.

[4] https://jinja.palletsprojects.com

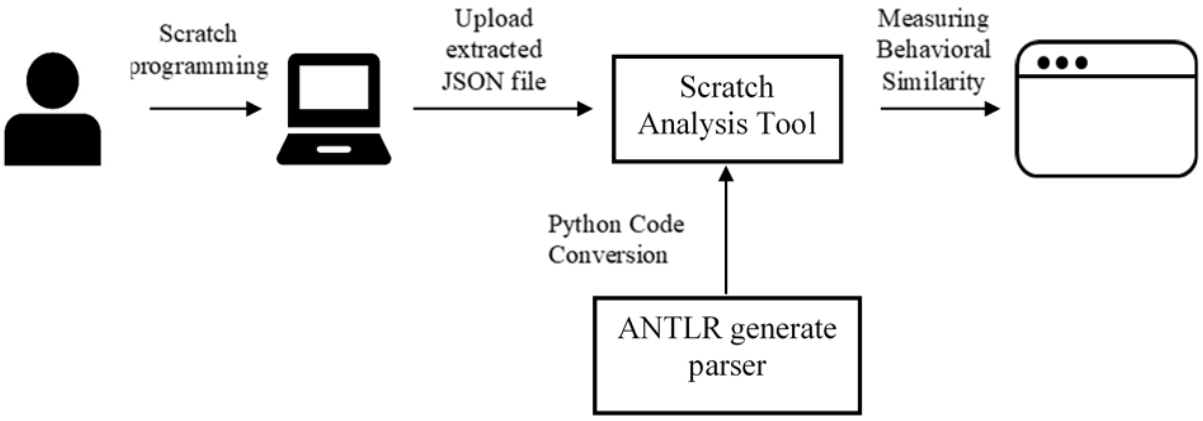


Figure 3: System module for analysis procedure

The prototype would allow its users to:

- Upload a JSON file(s) extracted from a Scratch project(s) that contains all the block-usage information;
- Mark one of the uploaded implementations as the solution to be used as reference;
- Delete uploaded projects together with all of its generated data by the tool;
- Calculate BS metrics;
- Modify the Python code and submit it in case of a syntax error;
- View log file generated during dynamic symbolic execution;
- View execution graph generated during dynamic symbolic execution;
- Examine code coverage report created using DSE generated inputs on Python code converted from JSON files;

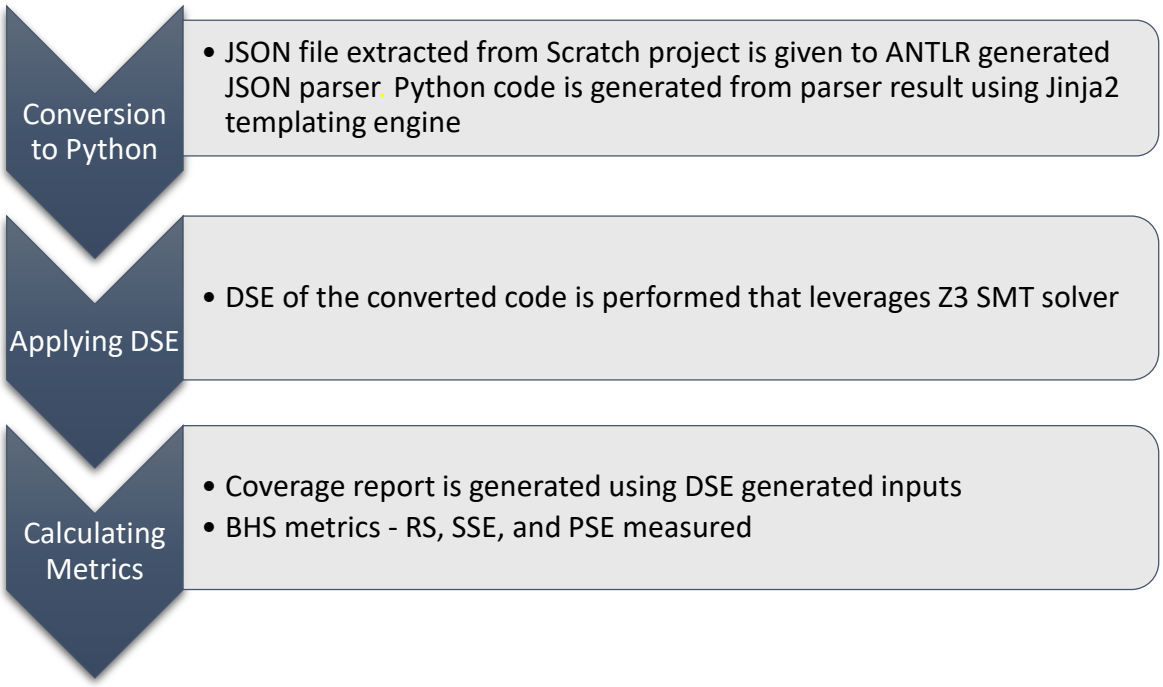


Figure 4. Steps performed by the tool grouped at three levels

The steps to generate metrics for BS, which are performed by the system in the order described in figure 4 are grouped at three levels - Conversion to Python, Applying DSE, Calculating Metrics. In the following sections each level is described in detail.

3.2.1 Converting Scratch program to Python code

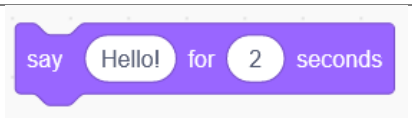
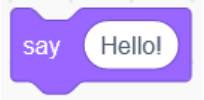
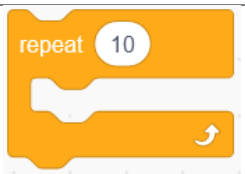
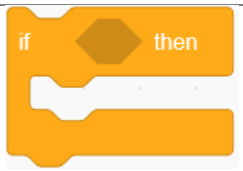
ANTLR, is a parser generator that can be used to read, execute, translate, or process binary files or structured text. It's commonly utilized in the creation of programming languages, frameworks, and tools[69]. ANTLR creates a language recognizer based on a formal grammar file containing customized lexical rules and syntax parser rules that show how a grammar matches the input. ANTLR can create Lexer and Parser code for various target languages, including C++, Java, JavaScript, and Python.

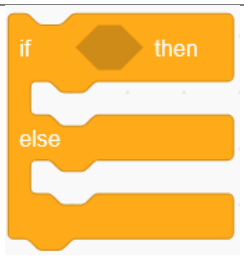
In order to convert Scratch code to its Python equivalent, initially, we tried regular expression, which proved highly inefficient as the converted code contained numerous errors. Fortunately, during the literature review, we discovered ANTLR, used for the same purpose in research[70] on Scratch analysis. Therefore, we adopted the same technique and set out to expand the JSON grammar file available in the official repository of the ANTLR project by adding 29 lexer rules

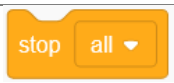
and 57 parser rules. It enabled mapping from 40 Scratch blocks to ANTLR parser rules listed in Table 5, which also shows their Python equivalents. Currently only the following categories of blocks are supported:

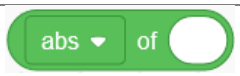
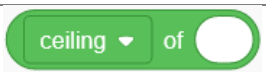
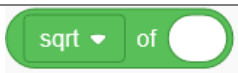
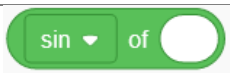
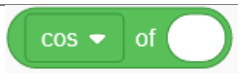
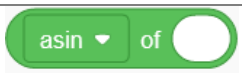
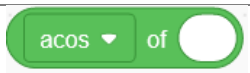
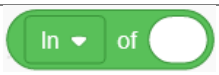
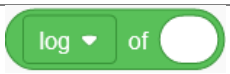
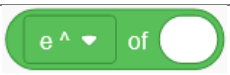
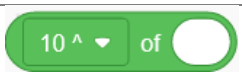
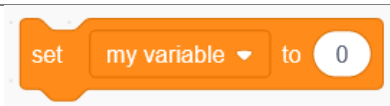
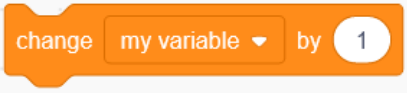
- Operators;
- Control (except “wait_seconds”, “wait_until”, “when_I_start”, “create_clone_of”, “delete_this_clone”);
- Looks (only “say_x_for_n_seconds”, “say_x” blocks);
- Variables (only “set_var_to_x”, “change_var_by_x” blocks);

Parts of the “project.json” file corresponding to other blocks not listed in table 5 are regarded as simple key-value pairs and ignored by the parser.

№	Visual block	Python equivalent
1		`return "Hello"`
2		`return "Hello"`
3		`for __index__ in range(10):` `  ...`
4		`while True:` `  ...`
5		`if condition:` `  ...`

6		`if condition:`
		`    ...`
		`else:`
		`    ...`

7		`while not condition:` `    ...`
8		[blank space]
9		`random.uniform(x, y)`
10		`(x > y)`
11		`(x < y)`
12		`(x and y)`
13		`(x or y)`
14		`(not condition)`
15		`"apple" + "banana"`
16		`"apple"[1 - 0]`
17		`len("apple")`
18		`("a" in "apple")`
19		`x % y`
20		`Decimal(x).to_integral_value(rou` `nding=ROUND_HALF_UP)`
21		`x * y`

22		`x - y`
23		`x + y`
24		`abs(x)`
25		`floor(x)`
26		`ceil(x)`
27		`sqrt(x)`
28		`sin(x)`
29		`cos(x)`
30		`tan(x)`
31		`asin(x)`
32		`acos(x)`
33		`tan(x)`
34		`log10(x)`
35		`log2(x)`
36		`exp(x)`
37		`pow(10, x)`
38		`variableName = 0`
39		`variableName =+ 1`

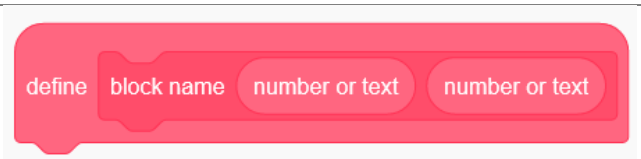
40		`def blockName(x, y):` `  ...`

Table 5. Supported Scratch blocks with their Python equivalent

Figure 5 shows the inner conversion flow of our prototype from a “look_sayforsecs” block to a corresponding Python code as an example. First, streams of chars from JSON are sequentially entered into the Lexer. The Lexer will next use lexical rules to take individual characters from the stream and turn them into tokens. The syntax parser then takes the token streams and transforms them into an organized formation by applying the syntax parser rules to recognize the sentence structure. Following lexical analysis and syntax parsing, an abstract syntax tree is generated (AST).

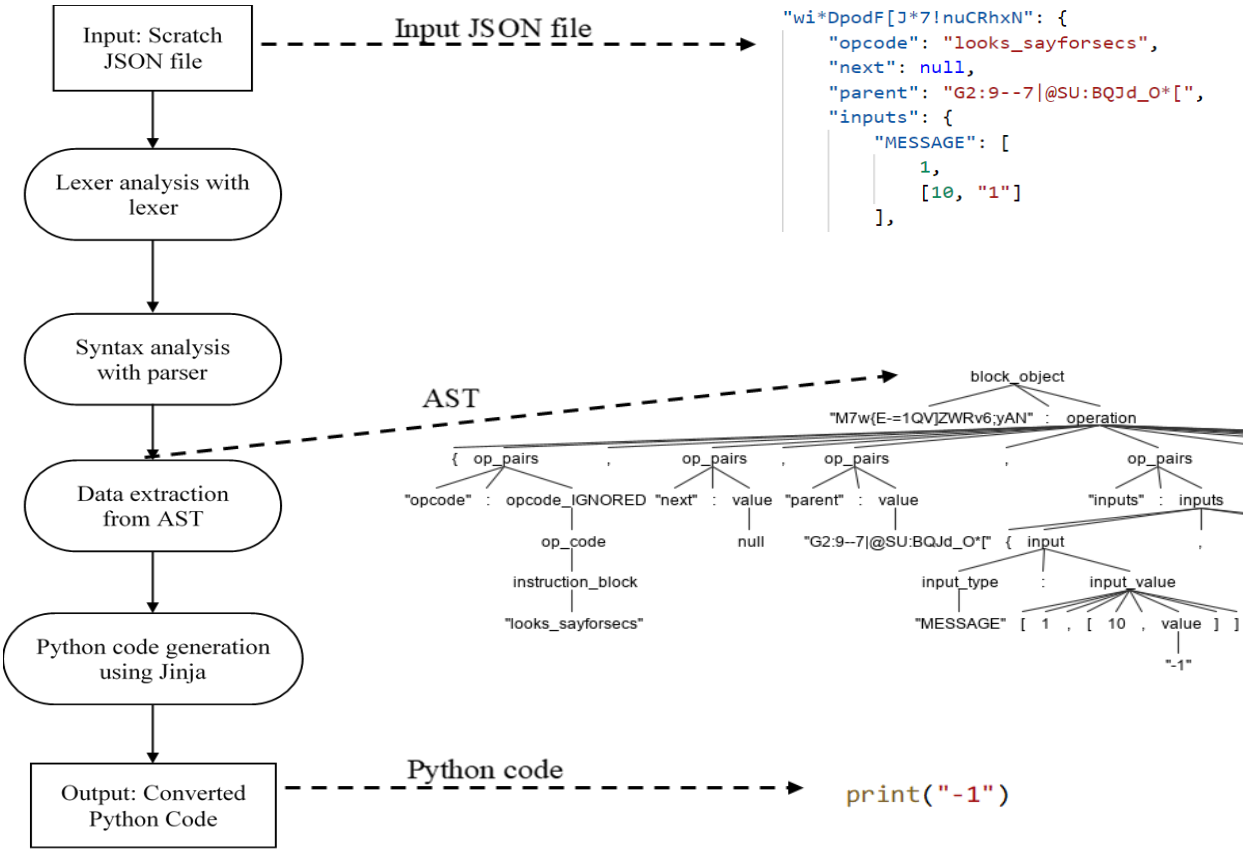


Figure 5. Overall conversion flow of Scratch code into Python

Analyzing the AST is an integral part of getting the final result (Python code). As it is difficult to edit and use tree structures directly, ANTLR uses a depth-first walk called Listener generated automatically to construct a tree walker navigating across the nodes of such trees to run application-specific code. In the Listener file, a pair of ANTLR generated enter/exit methods with default implementations exist for each parser rule we defined. Thus, it enables us to override desired methods in our derived Listener and use them for our need - to save data as inputs for template string literals.

As mentioned earlier, we used the Jinja2 template engine to generate Python code. When a specific code that matches the rule is detected, the enter/exit methods are invoked immediately. Data we are interested in is then extracted from the AST and stored in a variable. Then a Jinja2 template is passed variables to fill placeholders in template strings, rendering the pieces of final python code from Scratch blocks. Then the code is stored in a local JSON file named "storage.json" as part of a JSON object corresponding to a Scratch project.

3.2.2 Applying DSE

As mentioned earlier, we used PyExZ3 for applying DSE on code translated from Scratch projects for its convenience and easy use. It should be noted that PyExZ3 can only be used for methods that accept at least one argument, and it must return a non-void value. Furthermore, attributes can only be of type integers. Unfortunately, we have to limit the algorithmic problems that can be analyzed by our tool to a much smaller set due to these limitations. Having said so, algorithmic problems that deal with integers are pretty many. It does not necessarily have to be a math problem, either. In addition, all the control statements and loops in Python is supported, except infinite loops. Also, the project could be extended to resolve the current issues we are facing as future work. In order to get more detailed description of the architecture of PyExZ3 please refer to this work[42].

We searched for strategies to overcome these limitations. In order to get around the issues mentioned earlier, we considered representing characters of a string as integers. However, soon we realized that there is a problem - with PyExZ3, it is not possible to represent strings with variable lengths as integers. The reason is that the number of arguments given to a method must be invariable to generate inputs by PyExZ3. Based on our search online, it appeared that the majority of DSE tools are unable to work with Strings.

DSE is applied to uploaded JSON files immediately after they are translated into Python code. Generated inputs and outputs are then stored in a local JSON file named “storage.json” in a corresponding JSON object of the Scratch project (Figure 6).


```
"90c67ecc-c924-452b-b204-4f1f73a3982d": {
    "fileName": "solution",
    "entry": "lcm",
    "params": [
        "x",
        "y"
    ],
    "code": "from math import *   # pragma: no cover\nfrom decimal import *   # pragma:
    no cover\nimport random          # pragma: no cover\nimport math             #
    pragma: no cover\nimport sys             # pragma: no cover\n\ndef lcm(x, y):\n\tprod
    = ( x  *  y )\n\tif (prod == 0):\n\t\treturn 0\n\tif ( x  >  y ):\n\t\tgreater =  x
    \n\telse:\n\t\tgreater =  y \n\twhile True:\n\t\tif (((greater %  x ) == 0) and (
    (greater %  y ) == 0)):\n\t\t\tlcm = greater\n\t\t\treturn lcm\n\t\tgreater =
    (greater + 1)",
    "isReference": true,
    "inputs": [[0,0],[1,2],[3,2],[7,8],[8,2],[1,-16],[2,-3],[5,4],[3,-16],[8,15]],
    "outputs": [0,2,6,56,8,16,6,20,48,120],
```


Figure 6. JSON object containing data about a custom block definition

Figure 6 illustrates a JSON object inside the “storage.json” file generated by our prototype that contains data belonging to a custom Scratch block definition, which calculates the least common multiple of two integers. The highlighted input and output values are used to measure BS upon user request. In the next section, how metrics are calculated is discussed with an example.

3.2.3 Calculating Metrics

During literature review, we encountered Sihan Li, Xusheng Xiao and others’ work[25] that proposed three metrics – Single-program Symbolic Execution, Paired-program Symbolic Execution, Random Sampling, to measure behavioral similarity between programs written in C# programming language. These metrics have been explained thoroughly under the title “Application of symbolic execution in code analysis” in this work. We inspired by the study results and decided to adopt the metrics above for our prototype. To the best of our knowledge, before our work, no study proposing measurement for the behavioral similarity between programs created using a visual block-based programming language has been implemented. Additionally, our tool also

achieves high code coverage using inputs and outputs generated from DSE of Scratch projects. As mentioned earlier, these metrics are calculated using DSE generated input-output pairs. Hence these measurements can be obtained upon user request and only after the DSE of the uploaded projects complete. In addition, one of the uploaded Scratch projects must be marked as a reference before a user can request to get any of the BS metrics. Unlike BS metrics, code coverage can be measured for all projects, without specifying the reference implementation.

RS is calculated by randomly generating integers in the range of (-1000, 1000) from the normal distribution using NumPy[5], a Python library for working with arrays. The input values are generated for each argument and given to programs under analysis to compare the outputs. Although it provides a good approximation and is the most cost-effective method compared to SSE and PSE, with randomly generated inputs, the probability of getting interesting inputs that lead to differences in behaviors of programs is extremely low. Let us look at an example shown in Figure 7, which illustrates two slightly different implementations of compare methods accepting two arguments – x and y. These methods are converted into Python from Scratch projects. Let us assume that Figure 7a is the reference program, and 7b shows the submission.

```
1   def compare(a, b):
2       if ( a  >  b ):
3           return 1
4       else:
5           return "-1"
```

a) reference program

```
1   def compare(a, b):
2       if ( a  >  b ):
3           return 1
4       if ( a  <  b ):
5           return "-1"
6       return 0
```

b) submission

Figure 7 Compare method implementations converted from Scratch files

When we request RS of the submission, inputs are generated randomly using the "numpy.random.randint function", which is passed three default values for these parameters; "low" (-1000), "high" (1000), and "size" (100). Generated inputs are then passed into the two versions of the compare method to compare their results. The proportion of agreed inputs over the size of the generated inputs (100) determines RS, which, in this example, is 100%. However, the calculated

[5] https://numpy.org/

value does not accurately reflect BS between the two since, for instance, in (0, 0) values of inputs, the methods will output different results. SSE and PSE can be used to overcome the shortcomings of RS.

SSE is calculated using DSE generated inputs and outputs of the reference program by giving the inputs into the project under analysis and comparing outputs it produces with that of the reference. Thus, it helps to measure behavioral similarity or deviation of the tested program from the reference program. Furthermore, it can provide us with clues into students' understanding of a particular algorithm necessary to do the assigned task. It does so by uncovering interesting inputs that students failed to take into account.

Table 6 lists outputs of both programs – reference program (Figure 7a) and the submission (Figure 7b) when they are passed DSE generated inputs of reference program as attributes. The DSE generated inputs of the reference program are taken from the "storage.json" file, which acts as a database storing all the details of each uploaded Scratch program as a JSON object. As seen, for the input values of (0, 0), the student's submission deviates behaviorally from the reference program. SSE is calculated as the proportion of agreed inputs over all the inputs acquired during the DSE of the reference program. In this example, 1 / 2 = 0.5, thus SSE is 50%.

X	**y**	**Reference program**	**Submission**
0	0	-1	0
2	0	1	1

Table 6. Comparison of reference and submitted compare method implementations

Unlike SSE, PSE reflects the behavioral deviation of the submission from the reference and vice versa, as it includes input domains of both implementations. Therefore, it allows users to have a better idea of BS between programs.

PSE is calculated on a program created from a submission and the reference project using the template shown in Figure 8. The process of program pairing is triggered when a user mark one of the uploaded programs as a reference. When PSE is requested, the SMT solver explores the paired program and generates two sets of values for inputs and outputs. An output indicates if the programs under analysis yield the same value for a given input set.

```
1    def pairedProgram(*args):
2        return method1(args) == method2(args)
```

Figure 8. Template for paired program

Let us look at an example. Figure 9 illustrates a paired program created from two different implementations of compare methods (Figures 7a and 7b). For the input values of (0, 0), the paired program yields 0, meaning the compared programs behave differently. However, when it is passed (1, 1), it yields 1. Since the proportion of ones over all the outputs is 1/2 = 0.5, PSE is 50%.

```
1    import importlib
2
3    a_compare = importlib.import_module('static.reference').compare
4    b_compare = importlib.import_module('static.submission1').compare
5
6    def pairedProgram(a, b):
7        return a_compare(a, b) == b_compare(a, b)
```

Figure 9. paired program constructed from two compare method implementation

Based on the findings of [71], there is a high correlation between high grades and high code coverage. Also, as mentioned earlier, generally via DSE generated inputs, we can achieve high code coverage as the DSE of the program is continued until all the feasible paths are explored, or a preset code coverage is achieved. More detailed information about DSE is given under the title “Dynamic symbolic execution” in this thesis. Considering these facts, we decided to additionally add a feature to calculate code coverage and generate a code coverage report.

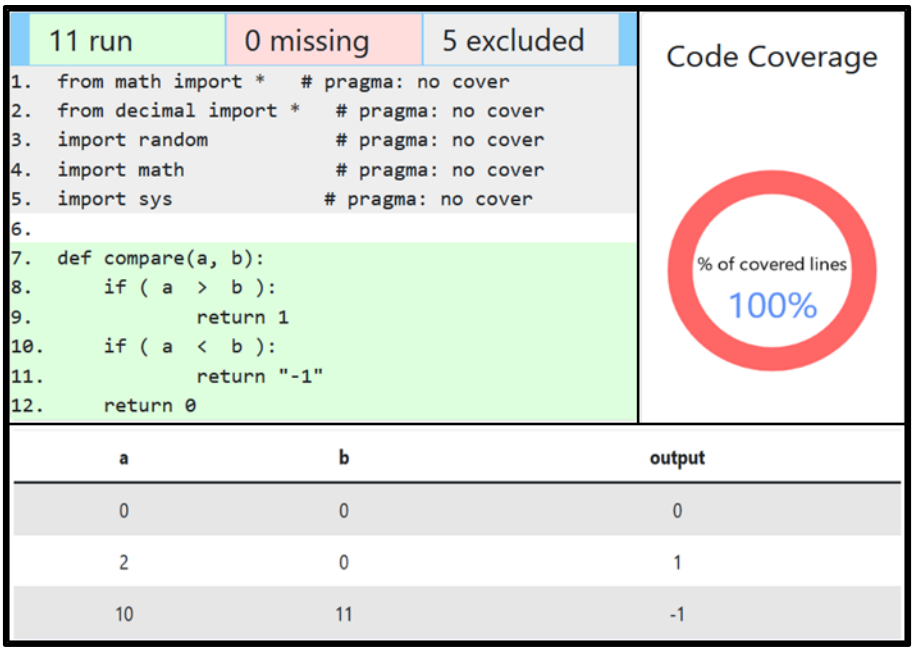


Figure 10. Code coverage report generated by our tool for compare method

When a user request code overage of a method, for instance, the compare method (Figure 7), its DSE generated inputs are extracted from the corresponding JSON object in the "storage.json" file. Then using Coverage.py API[6], code coverage of the method is measured, and the result is passed into the front-end side to create a coverage report shown in Figure 10. As seen, we achieved 100% code coverage using DSE generated input-output pairs listed in the coverage report. Note that the first five lines of the trasnpiled python code are highlighted with grey colour to indicate that they are excluded from the code coverage process. This is because they are not part of the compare method implementation and automatically added for each converted python code to be executed properly.

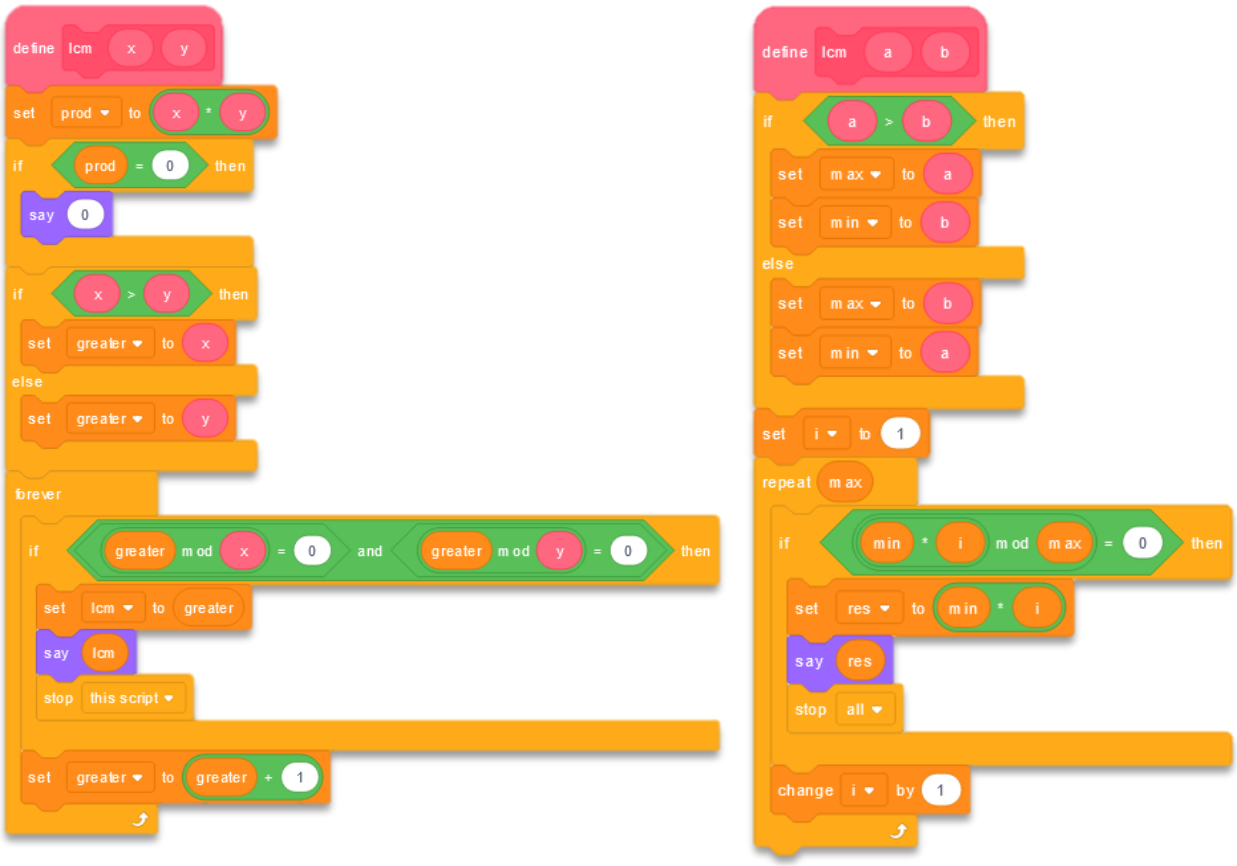
[6] https://coverage.readthedocs.io/en/coverage-5.5/api.html

4 Example

In this chapter, the tool usage has been explained step-by-step on an example, a custom block that finds out the least common multiple of two integers.

4.1 Least common multiple

In figure 11, we can see a two slightly different implementation of custom “lcm” block that calculates least common multiple (LCM) of two integers. Note that the names of the custom blocks that are compared must be the same. Let us consider figure 11a as the solution and 11b as a submission.



a) solution b) submission

Figure 11. Two version of Least common multiple implemented in Scratch

In order to analyze the two using our tool, first, “projects.json” files from both downloaded project files need to be extracted and uploaded to the system using the file form shown in Figure 12. Then, the “Function name” input field must be filled with the exact name of the custom blocks, which, in this case, is “lcm”.

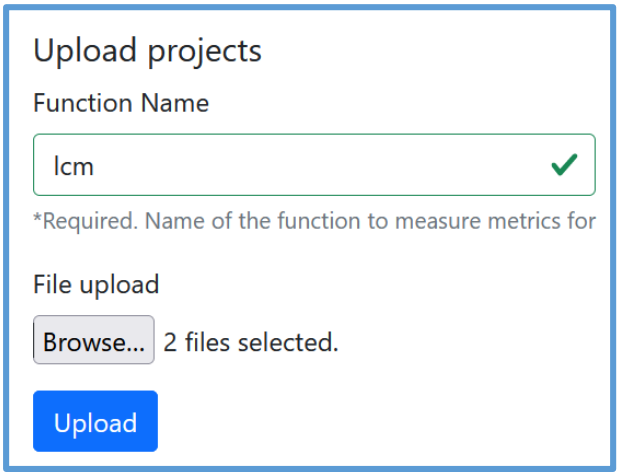


Figure 12. File Upload Form

The uploaded files (Figure 13) are immediately converted into Python code. Then, the tool generates input and output arrays for each of the two implementations by performing DSE on transpiled python code. At this point, it is possible to compute code coverage by clicking on the “CVG” button without marking any of the files as a reference program. In addition, we can delete an uploaded file by clicking on the “Del” button.

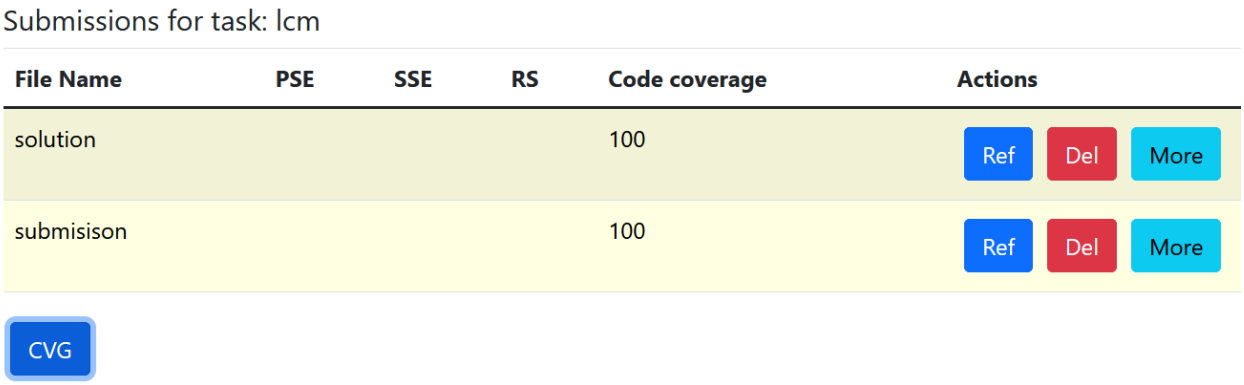


Figure 13. Scratch implementations uploaded to the system

As soon as one of the implementations is chosen as a reference by clicking on the “Ref” button, buttons (PSE, SSE, RS) to trigger measurements of BS metrics appear on the page (Figure 14).

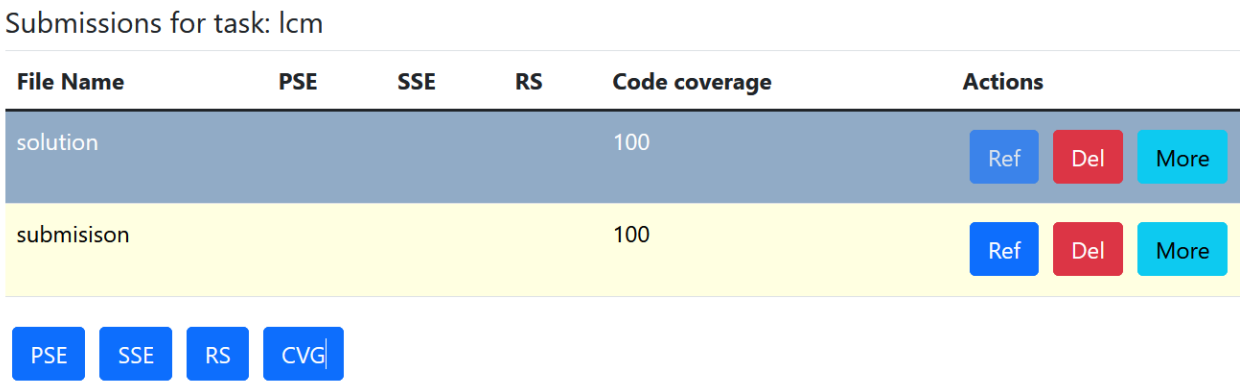


Figure 14. Table of submissions after a reference program is selected

Figure 15 illustrates the final results after making requests for all the metrics. In order to get more details about the analysis result of a particular submission, we can click the “More” button on the row.

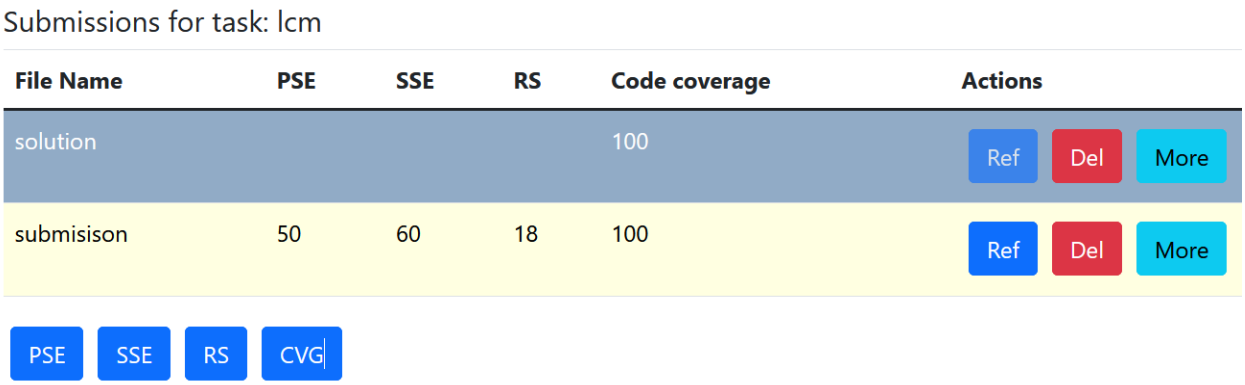


Figure 15. Calculated Metrics

Figure 16 shows the metrics in percentage as well as the input-output pairs used in calculating metrics. Behavioral deviation can be identified by examining the results. Based on the SSE metric,

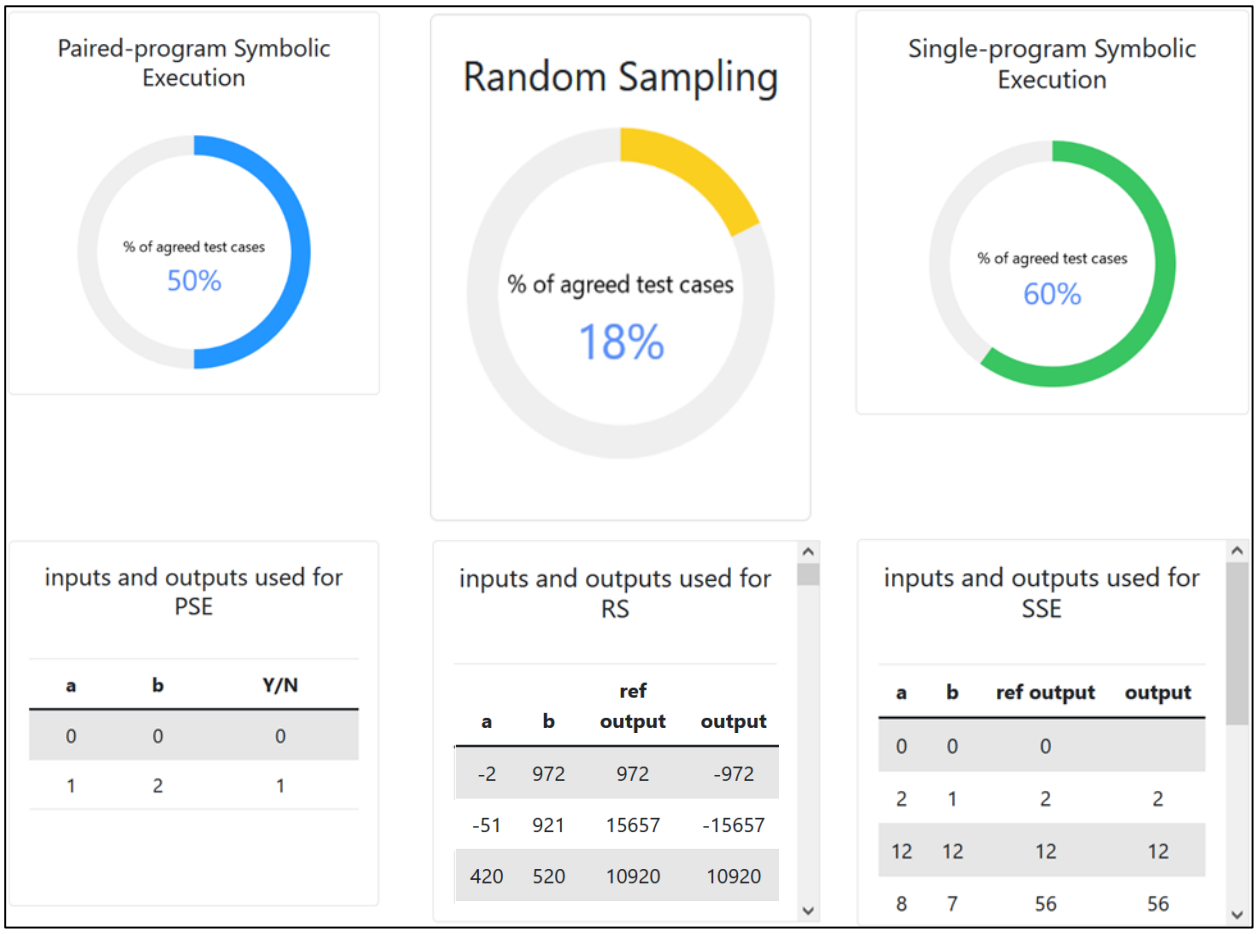
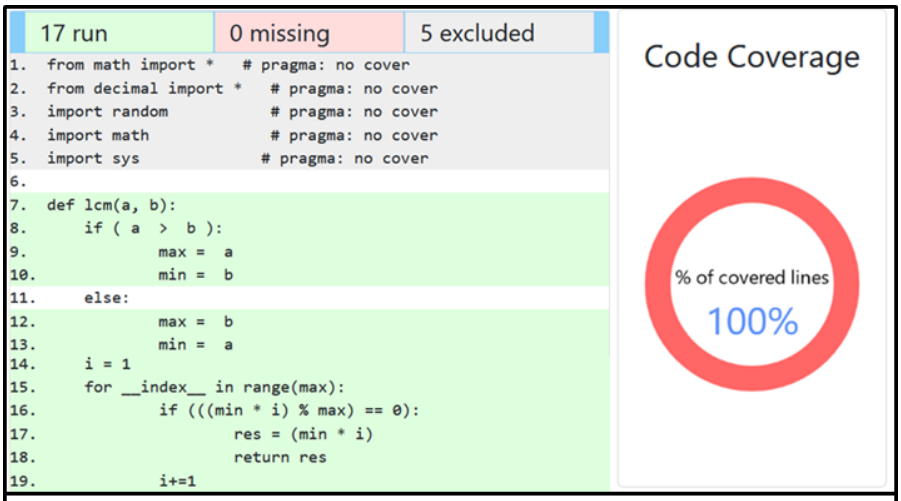


Figure 16 BS metrics with input output pairs

which is 60 per cent, we can say the submission deviates from the behavior of the reference program, which implies that the student is in the wrong direction. For instance, the input values of (0, 0) have not been considered during the submission implementation. Furthermore, regardless of the sign of integers given as input, the submission was expected to return a positive value as LCM represents the smallest positive integer that can be divisible by both integers. However, we see that the submission returns a negative value for negative inputs. We uncovered two cases a student has not considered, indicating their lack of understanding of LCM.




```
1.  from math import *   # pragma: no cover
2.  from decimal import *   # pragma: no cover
3.  import random          # pragma: no cover
4.  import math            # pragma: no cover
5.  import sys            # pragma: no cover
6.
7.  def lcm(a, b):
8.      if ( a  >  b ):
9.              max =  a
10.             min =  b
11.     else:
12.             max =  b
13.             min =  a
14.     i = 1
15.     for __index__ in range(max):
16.             if (((min * i) % max) == 0):
17.                     res = (min * i)
18.                     return res
19.             i+=1
```

a	b	output
0	0	
2	0	0
2	8	8
-12	-16	
8	1	8
8	8	8
4	8	8
2	6	6
4	15	60
8	4	8

Figure 17. Code coverage report with a list of DSE generated input-output pairs

As can be seen from Figure 17, we achieved a high code coverage thanks to DSE generated inputs, which is a good sign indicating no unreachable paths is detected. In addition, we can modify the translated Python code, which could be helpful to see how much a mistake can affect a calculated metric (Figure 18). For instance, in submission, the student forgot to return the absolute value of the “res” variable.

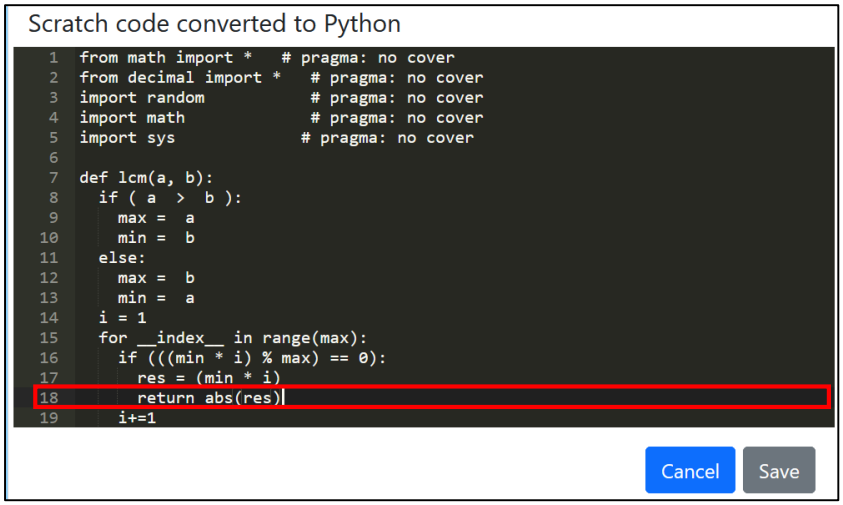


Figure 18. Code editor

After making the modification highlighted with red rectangular in Figure 18, the RS rose from 18% to 78%.

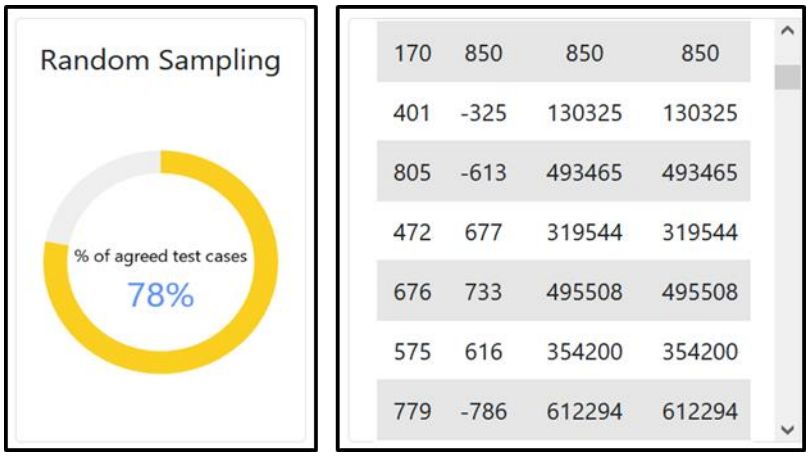


Figure 19. RS after minor modification illustrated in Figure 18

The analysis result helps to find out behavioral differences between the programs. Furthermore, it helps identify edge cases that have not been taken into account by the program's author and provides us with insights into the students' understanding of the algorithm.

5 Conclusion

In this thesis work, we described the implementation of DSEScratch, a web tool that measures four metrics, namely: Code Coverage, RS, SSE, and PSE using DSE, which previously have not been implemented in the context of visual block-based programming languages. However, due to the limitations of underlying technologies, the range of Scratch programs that our system can analyse is reduced to a much smaller set of programs. Programs that can be analysed by our tool must contain one custom block definition made of only a subset of Scratch blocks, which must accept at least one integer argument and must return an integer or a string value. Nevertheless, it demonstrates a novel way of analysing Scratch code for evaluation purposes.

Sihan Li, X. Xiao and others[25] suggested several applications of BS; none of them was related to analysing tasks to assess CT skills. However, based on our assumptions, we thought they could be used to gain insights into the CT skills of students. We have tried to explain how this could be useful in an example. We believe edge cases that students have not considered during implementing submissions speak about their understanding of a task or an algorithm. Furthermore, as unit-testing, BS can indicate functional correctness if we assume that the reference program is 100% correct. However, unlike conventional unit testing, the inputs are generated automatically based on the referral program or randomly. It means the need for manual labour is decreased significantly using this tool, and more submissions can be analysed.

Thus, this tool provides us with BS metrics that can be used to identify gaps in students' understanding of algorithms or check functional correctness. On the other hand, Code coverage helps us see any unreachable code in submissions that illustrates the quality of the students' code. In addition, according to a study result[26], there is a correlation between high code coverage and higher grades assigned to students.

Nevertheless, there are several functionalities we would like to add to the tool. For instance, currently, our system is only meant to be used by instructors. However, the tool could be extended for students to exercise or train their CT skills by creating or modifying block-based projects and submitting them at the same place without using third-party systems. For this purpose, we could use Blocky- A project of Google, free and open-source software for creating visual block-based programming languages and editors, which resembles Scratch.

Another feature is to use BS as a hint to show students their progress, which would motivate them to develop further or improve their implementations. In addition, as a hint, failed test cases

can be shown to students on request – information to motivate them to examine the code and exercise their debugging and problem-solving skills.

We would also like to add a feature to track user actions, such as the number of times a student attempted to submit his/her implementation, the kind of blocks used in a project. We believe it would help instructors better learn how students understand a particular algorithm or identify their weaknesses or which CT concept they are struggling to comprehend.

References

[1] *K-12 Computer Science Framework*. .

[2] J. Hattie and H. Timperley, “The Power of Feedback,” vol. 77, no. 1, pp. 81–112, 2007.

[3] A. Groce and W. Visser, “What Went Wrong : Explaining Counterexamples.”

[4] V. J. Shute, “Focus on Formative Feedback,” vol. 78, no. 1, pp. 153–189, 2008.

[5] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*. 1993.

[6] J. Wing, “Computational Thinking,” *Commun. ACM*, vol. 49, pp. 33–35, 2006.

[7] F. Kalelioglu, Y. Gulbahar, and V. Kukul, “A Framework for Computational Thinking Based on a Systematic Research Review,” no. May, 2016.

[8] R. Millwood, N. Bresnihan, and D. W. and J. Hooper, “Primary Coding Review of Literature on Computational Thinking,” *Natl. Counc. Curric. Assess.*, 2018.

[9] D. Hickmott, E. Prieto-Rodriguez, and K. Holmes, “A Scoping Review of Studies on Computational Thinking in K–12 Mathematics Classrooms,” *Digit. Exp. Math. Educ.*, vol. 4, no. 1, pp. 48–69, 2018.

[10] M. Román-González, J. Moreno-León, and G. Robles, “Combining Assessment Tools for a Comprehensive Evaluation of Computational Thinking Interventions,” 2019, pp. 79–98.

[11] K. Brennan and M. Resnick, “New frameworks for studying and assessing the development of computational thinking,” in *American Educational Research Association*, 2012.

[12] A. Aho, “Computation and Computational Thinking,” *Comput. J.*, vol. 55, pp. 832–835, 2012.

[13] J. M. Wing, “Computational Thinking : What and Why ?,” no. November, pp. 1–6, 2010.

[14] H. Hoppe and S. Werneburg, “Computational Thinking—More Than a Variant of Scientific Inquiry!,” 2019, pp. 13–30, + supplement.

[15] and L. C. David Barr, John Harrison, “Computational Thinking: A Digital Age Skill for Everyone,” 2011. [Online]. Available: moz-extension://fe4ce275-77d0-48b3-a470-

4a2958db111d/enhanced-reader.html?openApp&pdf=https%3A%2F%2Fid.iste.org%2Fdocs%2Flearning-and-leading-docs%2Fmarch-2011-computational-thinking-ll386.pdf. [Accessed: 07-Feb-2020].

[16] S. Lye and J. Koh, "Review on teaching and learning of computational thinking through programming: What is next for K-12?," *Comput. Human Behav.*, vol. 41, pp. 51–61, 2014.

[17] M. Syslo and A. Kwiatkowska, "Informatics for all high school students: A computational thinking approach," 2013, vol. 7780, pp. 43–56.

[18] J. Pellegrino and M. Hilton, "Education for Life and Work: Developing Transferable Knowledge and Skills in the 21st Century," 2012.

[19] S. Kong, "Components and Methods of Evaluating Computational Thinking for Fostering Creative Problem-Solvers in Senior Primary School Education," 2019, pp. 119–141.

[20] C. Whittaker, S. Salend, and D. Duhaney, "Creating Instructional Rubrics for Inclusive Classrooms," *Teach. Except. Child.*, vol. 34, pp. 8–13, 2001.

[21] R. Mccauley, "Rubrics as assessment guides," *SIGCSE Bull.*, vol. 35, pp. 17–18, 2003.

[22] S. Grover, S. Cooper, and R. Pea, "Assessing computational learning in K-12," *ITICSE 2014 - Proc. 2014 Innov. Technol. Comput. Sci. Educ. Conf.*, 2014.

[23] B. Zhong, Q. Wang, J. Chen, and Y. Li, "An Exploration of Three-Dimensional Integrated Assessment for Computational Thinking," *J. Educ. Comput. Res.*, vol. 53, 2015.

[24] Q. Burke, "The Markings of a New Pencil: Introducing Programming-as-Writing in the Middle School Classroom," *J. Media Lit. Educ.*, vol. 4, pp. 121–135, 2012.

[25] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann, "Measuring code behavioral similarity for programming and software engineering education," *Proc. - Int. Conf. Softw. Eng.*, pp. 501–510, 2016.

[26] A. Stahlbauer, M. Kreis, and G. Fraser, "Testing Scratch Programs Automatically."

[27] D. E. Johnson, "ITCH: Individual testing of computer homework for scratch assignments," *SIGCSE 2016 - Proc. 47th ACM Tech. Symp. Comput. Sci. Educ.*, pp. 223–227, 2016.

[28] F. Maiorana, D. Giordano, and R. Morelli, “Quizly: A live coding assessment platform for App Inventor,” *Proc. - 2015 IEEE Blocks Beyond Work. Blocks Beyond 2015*, no. October, pp. 25–30, 2015.

[29] M. G. Vera Barstad Terje Gjøsæter , , Faculty of Engineering and Science, University of Agder, Serviceboks 509, NO-4898 Grimstad, Norway, vera.barstad@gmail.com, {morten.goodwin, terje.gjosater}@uia.no, “Predicting Source Code Quality with Static Analysis and Machine Learning Vera Barstad, Morten Goodwin, Terje Gjøsæter,” no. 4898, pp. 1–12, 2014.

[30] N. Truong, P. Roe, and P. Bancroft, “Static Analysis of Students’ Java Programs.,” 2004, pp. 317–325.

[31] J. S. Song, S. H. Hahn, K. Y. Tak, and J. H. Kim, “An intelligent tutoring system for introductory C language course,” *Comput. Educ.*, vol. 28, no. 2, pp. 93–102, 1997.

[32] D. FONTE, D. da Cruz, A. Gançarski, and P. Rangel Henriques, “A flexible dynamic system for automatic grading of programming exercises,” *OpenAccess Ser. Informatics*, vol. 29, 2013.

[33] F. Alshamsi and A. Elnagar, “An automated assessment and reporting tool for introductory Java programs,” *2011 Int. Conf. Innov. Inf. Technol. IIT 2011*, 2011.

[34] D. da Cruz, I. Boas, D. FONTE, A. Gançarski, and P. Rangel Henriques, “Program analysis and evaluation using QUIMERA,” *ICEIS 2012 - Proc. 14th Int. Conf. Enterp. Inf. Syst.*, vol. 2, 2012.

[35] T. Wang, X. Su, P. Ma, Y. Wang, and K. Wang, “Ability-training-oriented automated assessment in introductory programming course,” *Comput. Educ.*, vol. 56, pp. 220–226, 2011.

[36] M. Joy, N. Griffiths, and R. Boyatt, “The BOSS Online Submission and Assessment System,” *ACM J. Educ. Resour. Comput.*, vol. 5, 2005.

[37] T. Zhang, P. Wang, and X. Guo, “A survey of symbolic execution and its tool kleE,” *Procedia Comput. Sci.*, vol. 166, pp. 330–334, 2020.

[38] R. Boyer, B. Elspas, and K. Levitt, “SELECT—A formal system for testing and debugging programs by symbolic execution,” *ACM SIGPLAN Not.*, vol. 10, pp. 234–245, 1975.

[39] W. Howden, “Symbolic testing and the DISSECT symbolic evaluation system,” *Softw. Eng. IEEE Trans.*, vol. SE-3, pp. 266–278, 1977.

[40] J. King, “A New Approach to Program Testing.,” in *ACM SIGPLAN Notices*, 1974, vol. 10, pp. 278–290.

[41] J. King, “Symbolic Execution and Program Testing,” *Commun. ACM*, vol. 19, pp. 385–394, 1976.

[42] T. BALL, J. DANIEL, and T. Ball, “Deconstructing Dynamic Symbolic Execution,” IOS Press, Jan. 2015.

[43] L. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, vol. 4963, pp. 337–340.

[44] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, vol. 56, pp. 82–90, 2013.

[45] R. Baldoni, E. Coppa, D. C. D. Elia, and C. Demetrescu, “A Survey of Symbolic Execution Techniques,” vol. 51, no. 3, 2018.

[46] N. Da Cruz Alves, C. Gresse Von Wangenheim, and J. C. R. Hauck, “Approaches to assess computational thinking competences based on code analysis in K-12 education: A systematic mapping study,” *Informatics Educ.*, vol. 18, no. 1, pp. 17–39, 2019.

[47] M. Mut-Puigserver, M. Magdalena Payeras-Capellá, J. Castellá-Roca, and L. Huguet-Rotger, “mCITYPASS: Privacy-preserving secure access to federated touristic services with mobile devices,” *Stud. Comput. Intell.*, vol. 727, pp. 135–160, 2018.

[48] B. Wolz, U., Hallberg, C., Taylor, “Scrape: A tool for visualizing the code of scratch programs,” in *Proc. of the 42nd ACM Technical Symposium on Computer Science Education*, 2011.

[49] B. Boe *et al.*, “Hairball : Lint-inspired Static Analysis of Scratch Projects,” 2013.

[50] J. Moreno-león, U. Rey, and J. Carlos, “Dr . Scratch : a Web Tool to Automatically Evaluate

Scratch Projects Dr . Scratch : a Web Tool to Automatically Evaluate Scratch Projects," no. October, pp. 10–13, 2015.

[51] G. Birch, B. Fischer, and M. Poppleton, "Fast test suite-driven model-based fault localisation with application to pinpointing defects in student programs," *Softw. Syst. Model.*, vol. 18, no. 1, pp. 445–471, 2019.

[52] H. Duong, T. Nguyen, and S. Chandra, "SemFix : Program Repair via Semantic Analysis," pp. 772–781, 2013.

[53] H. Pham, *Software Reliability*. Berlin: Springer, 2000.

[54] M. Ahmadzadeh, D. Elliman, and C. Higgins, "An Analysis of Patterns of Debugging Among Novice," no. February 2014, 2005.

[55] I. Vessey, "Expertise in debugging computer programs : A process analysis," pp. 459–494, 1985.

[56] S. Chandra, U. C. Berkeley, and U. C. Berkeley, "Angelic Debugging," *2011 33rd Int. Conf. Softw. Eng.*, pp. 121–130, 2011.

[57] A. Cleve, H., Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, 2005, pp. 342–351.

[58] A. Griesmayer, S. Staber, and R. Bloem, "Automated Fault Localization for C," *Electron. Notes Theor. Comput. Sci.*, vol. 174, no. 4, pp. 95–111, 2007.

[59] A. Groce, "Error Explanation with Distance Metrics," no. Figure 1.

[60] M. Jose and R. Majumdar, "Cause Clue Clauses : Error Localization using Maximum Satisfiability."

[61] M. Renieris and S. P. Reiss, "Fault Localization With Nearest Neighbor Queries," 2003.

[62] J. Criswell, "Using Likely Invariants for Automated Software Fault Localization," pp. 139–151.

[63] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight Fault-Localization Using Multiple Coverage Types," *2009 IEEE 31st Int. Conf. Softw. Eng.*, pp. 56–66, 2009.

[64] N. Tillmann, J. D. Halleux, T. Xie, and J. Bishop, “Pex4Fun: A web-based environment for educational gaming via automated test generation,” *2013 28th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pp. 730–733, 2013.

[65] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. D. Halleux, “Code Hunt: Experience with Coding Contests at Scale,” *2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 2, pp. 398–407, 2015.

[66] B. Loring, D. Mitchell, and J. Kinder, “ExpoSE: practical symbolic execution of standalone JavaScript,” 2017, pp. 196–199.

[67] G. Li, E. Andreasen, and I. Ghosh, “SymJS: automatic symbolic testing of JavaScript web applications,” 2014, pp. 449–459.

[68] N. Tillmann and P. de Halleux, “Pex - White Box Test Generation for .NET,” in *Proc. of Tests and Proofs (TAP’08)*, 2008, Proc. of T., vol. 4966, pp. 134–153.

[69] T. Parr, S. Harwell, and K. Fisher, “Adaptive LL(*) Parsing: The Power of Dynamic Analysis,” *SIGPLAN Not.*, vol. 49, no. 10, pp. 579–598, Oct. 2014.

[70] Z. Chang, Y. Sun, T. Y. Wu, and M. Guizani, “Scratch Analysis Tool(SAT): A Modern Scratch Project Analysis Tool based on ANTLR to Assess Computational Thinking Skills,” *2018 14th Int. Wirel. Commun. Mob. Comput. Conf. IWCMC 2018*, pp. 950–955, 2018.

[71] A. Stahlbauer, M. Kreis, and G. Fraser, “Testing scratch programs automatically,” *ESEC/FSE 2019 - Proc. 2019 27th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, pp. 165–175, 2019.

License

Non-exclusive license to reproduce thesis and make thesis public

I, Ismat Alakbarov

1. herewith grant the University of Tartu a free permit (non-exclusive license) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,
   Dynamic Analysis of Scratch Projects to Infer Computational Thinking Abilities
   supervised by Marcello Sarini and Marlon Dumas
2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons license CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Ismat Alakbarov

15/07/2021