Discriminating Programming Strategies in Scratch - Making the Difference between Novice and Experienced Programmers

Max Kesselbacher
max.kesselbacher@aau.at
Universität Klagenfurt
Klagenfurt, Austria

Andreas Bollin
andreas.bollin@aau.at
Universität Klagenfurt
Klagenfurt, Austria

ABSTRACT

Nowadays, block-based programming environments are often used to offer a gentle introduction to learning a programming language. However, an assessment of students' programming skills based on the results of a programming task is not sufficient to determine all areas students are struggling with. We therefore introduce a learning analytics approach of measuring and evaluating the programming sequences of students that program with Scratch 3. With our measurement framework, it is possible to record, store and analyze programming sequences done on a publicly-available, instrumented Scratch 3 environment. Changes in the programming sequence are categorized regarding the used block types and types of program change. We conducted an exploratory programming trial with lower and upper secondary school students to investigate small-scale programming strategies in the recorded programming sequences. Our goals are to identify students in need of support and to identify recurring patterns used by students successful in the trial. Clustering with k-means makes it possible to identify struggling students based on both interacted block types and types of program changes. Recurring patterns in the programming sequences of successful students show that small-scale programming strategies are very diverse.

CCS CONCEPTS

• **Social and professional topics → Student assessment**; *Computational thinking*; *Software engineering education*; *K-12 education*; • **Software and its engineering →** *Software creation and management*.

KEYWORDS

block-based programming, learning analytics, programming patterns

ACM Reference Format:
Max Kesselbacher and Andreas Bollin. 2019. Discriminating Programming Strategies in Scratch - Making the Difference between Novice and Experienced Programmers. In *14th Workshop in Primary and Secondary Computing Education (WiPSCE'19), October 23–25, 2019, Glasgow, Scotland Uk.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3361721.3361727

1 INTRODUCTION

Algorithmic thinking is part of the skill set of computational thinking, which is currently being implemented in curricula and educational standards all around the world. Algorithmic thinking is not only exercised by programming - Bollin et al. [9] have found that Bebras tasks offer a variety of problems and manage to inspire pupils with similar traits compared to programming students. Still, programming is a key skill of computer science, and basic concepts should be learned by everybody.

The fail rates of introductory programming courses show that learning to program continues to be a hurdle for students [38]. Block-based programming environments like Scratch [27] offer advantages that make them well-suited to introduce programming, even at an early age [32]. Armoni et al. [3] demonstrate that Scratch is a suitable first-exposure programming environment and can improve the acquisition of text-based programming skills. Grillenberger and Romeike [14] show that advanced computer science concepts can be taught to upper secondary school students with the help of block-based programming environments. But Hermans and Aivaloglou [17] also show that code smells found in Scratch projects hamper students' learning progress.

Traditionally, students' programming skills are assessed based on an artifact evaluation after task completion. Even when employing a periodic assessment based on an automated evaluation (like Troiano et al. [37] use *Dr. Scratch* [30]), drawbacks of these assessments are that the student's learning opportunity is already over and that bad habits which lead to code smells may have already occurred.

Learning analytics approaches [19] are well-suited to overcome the drawbacks of after-task assessment. The objective of this paper is to discriminate students based on their programming sequences, with the aim of evaluating a student's progress during task completion and improving instructions on an individual basis. As a step towards our aim, we thus introduce an IDE-based measurement framework for Scratch 3 which makes it possible to record, store and analyze students' programming sequences when working on a task. From the collected data, we identify metrics that discriminate between students in need of support and successful students. We address the following research questions:

RQ1 How can block-based programming sequence data from an IDE-based learning analytics setting be used to identify students in need of support?
RQ2 What programming sequence metrics and recurring patterns discriminate students successful in the trial?

To obtain answers to these research questions, we measured the program construction of four cohorts ($n = 42$) of students aged 13 – 18, solving a given programming task in Scratch 3. We categorized the students' program changes regarding the used block types and types of program change. We analyzed the data with correlation and cluster analysis, and extracted small-scale recurring patterns in the programming sequences. We contribute the following to the body of knowledge:

- An IDE-based learning analytics framework to record programming sequences in Scratch 3.
- A set of categories to classify programming sequences in Scratch 3, based on recorded learning analytics data.
- Identification of programming sequence metrics that discriminate students in need of support from successful students.

2 RELATED WORK

Block-based programming languages like Scratch [27] and Code.org [10] are widely used in today's school classrooms to expose students to programming for the first time. There has already been effort to investigate the use of block-based programming languages for teaching computer science related topics and programming in particular. The group of Armoni, Mordechai and Meerbaum-Salant has published a number of articles on this topic. Chronologically, first Meerbaum-Salant et al. [25] describe certain programming *habits* that are fostered by learning programming in Scratch and are contradictory with accepted software engineering practices: a bottom-up development process, and *extremely fine-grained programming* where solutions are programmed in a fine-grained, not a general way. Later, Meerbaum-Salant et al. [26] describe a taxonomy, incorporating both SOLO [6] and Bloom's [2] taxonomies, to measure the learning of computer science concepts with environments like Scratch. Armoni et al. [3] conducted an experiment in which they investigate the effects of transitioning from Scratch programming to textual programming (C# or Java) in secondary schools, with regards to the acquired programming knowledge. They found that students familiar with Scratch needed less time to learn new topics, had fewer learning difficulties and achieved a higher cognitive level of understanding. At the end of the teaching process, there were no significant differences in the achievements, regardless of Scratch familiarity.

These results make a point in favor of teaching programming with learning environments like Scratch. But the articles also show that care must be taken to prepare the correct instructions for learning how to program. Other articles investigated specific learning interactions with the Scratch environment. Swidan et al. [35] examined how Scratch programmers name their variables and procedures and found that they prefer longer identifier names compared to names used in textual programming languages. More than one third of the analyzed projects use identifier names that contain spaces, a feature unique to Scratch as opposed to textual programming languages. The authors argue that this can hamper the transition from Scratch to a textual programming language. Grover and Basu [15] describe an experiment made with middle school students and report on misconceptions of loops, variables and boolean logic used in Scratch programs. The authors show that, even after completing an introductory programming course with Scratch, the students are unfamiliar with the use of variables, and have misconceptions with how loops and boolean logic operators work.

Also the presence of code smells in block-based program scripts has been examined. Hermans et al. [18] report that more than 88% of the analyzed projects contain code smells, most frequently *lazy class*, *duplication*, and *dead code* smells. Hermans and Aivaloglou [17] conducted a controlled experiment with novice Scratch programmers to investigate the effects of code smells when comprehending Scratch programs. They found that *long method* code smells lead to a decreased system understanding, while *duplication* code smells lead to a decreased ease of program modification. These results imply that code smells already have an effect on Scratch program scripts. When transitioning students to a textual programming language, care must be taken to prevent the transfer of program construction strategies that lead to more *smelly* program code.

To adapt teaching instructions with this goal in mind, the measurement of patterns and strategies used during program construction can provide additional insight. Meerbaum-Salant et al. [25] describe programming *habits* but do not quantify the effects with measurable program interactions. Boe et al. [8], Moreno et al. [30], and Aivaloglou and Hermans [1] perform static analysis of Scratch code repositories, but do not consider dynamic program construction. In the context of textual programming, programming patterns have already been the focus of different researchers. Already twenty years ago, Kontogiannis [22] investigated whether patterns can be detected in source code with the use of software metrics. The author focused on finding duplicated code fragments by using structural and data flow metrics. Proulx [33] considered the incorporation of design and programming patterns into introductory computer science courses, acknowledging that there are patterns in programming that are beneficial to convey to students.

Since then, the field of *learning analytics* or *educational data mining* has steadily grown. Researchers make use of the collection and analysis of various data in educational processes in order to gain empirical support for educational theories on a completely new scale [4]. In text-based programming, patterns in clustered program states have been investigated by Blikstein et al. [7] to identify struggling students and to predict student achievement. Rivers et al. [34] employed a fine-grained assessment of programming skills by analyzed students' learning curves regarding used syntax elements in Python.

Block-based programming and Scratch in particular have already been studied with *learning analytics*. Papavlasopoulou et al. [31] statically analyze Scratch projects regarding the used programming concepts by mapping them to Scratch blocks. They report strong correlations between the use of threads and looping, as well as variables and event handling, looping, threads and operators. Filvà et al. [12] describe a learning analytics infrastructure based on instrumenting Scratch 2, storing the click stream of programming actions and reporting behaviour patterns to instructors. They extract two types of behaviour patterns from an experimental cohort: execution behaviour, and coding trend detected by click concentration. Kesselbacher and Bollin [21] describe an experiment in Scratch 2, recording students' programming sequences with click, keyboard and screenshot logging. They found that students' programming skills highly correlate with the change rate of the used types, and identified four patterns and strategies (*Trial & Error*, *Unfamiliarity*, *Late Abstraction* and *Subprogram* construction) with their characterizing metrics using association rules mining.

We extend the related work on *learning analytics* in Scratch by introducing an IDE-based measurement framework for Scratch 3. Our focus is the analysis of programming sequences instead of click streams, and we show that metrics obtained from the programming sequence can be used to discriminate between successful students and those who struggle. Unlike an after-task assessment, our approach can be used to individually support students during programming.

In addition to the above approaches, we use the neo-Piagetian theory to put our findings into context. Lister has extensively studied the first three stages of neo-Piagetian development (*sensorimotor*, *preoperational*, *concrete-operational*) with regards to novice programmers' development of programming skills [23]. In Lister's work, those stages are called *pre-tracing*, *tracing*, and *post-tracing* and describe the novice's ability to mentally trace and reason about programming code [24]. We use these terms in our work.

3 METHODOLOGY

In this section, we first describe the IDE-based learning analytics measurement framework for Scratch 3. Next, the trial example and the measured student cohorts of the programming trial are detailed. Lastly, we describe the data collected by the measurement framework and used in analysis and discussion of the results.

3.1 Measurement Framework for Scratch 3

The proposed measurement framework[1] is implemented as a custom online Scratch 3 environment, and represents an IDE-based learning analytics framework following Hundhausen et al. [19]. The framework provides an unobtrusive way to record programming sequences and requires no specific interaction of the user.

The source code is instrumented on server side to record users' programming interactions (mouse and keyboard events), users' performed changes to program blocks and a representation of the whole program script after each change. The instrumented IDE makes it possible to record fine-grained changes to the abstract syntax tree (AST) representing the executable Scratch program. The state of the AST is maintained by the Scratch virtual machine [29] by listening for changes and observing block change events. The event types are emitted by the Scratch blocks framework [28], a fork of Google's Blockly project [13], and constitute the category of block listen events in Table 2. See Section 3.3 for details regarding the classification of performed changes.

Following the classification of data collection for learning analytics in programming (Ihantola, Vihavainen et al. [20]), this measurement framework records programming data by *IDE instrumentation* and *key logging*. Regarding the granularity, each programming interaction in Scratch produces a compilable and executable state in the underlying virtual machine. The measurement procedure records data on *key stroke* and *compilation* granularity.

3.2 Programming Trial

The programming trial, used already in our previous work [21], consists of a given programming task, which was crafted to require multiple programming concepts: loops, conditional branching, variables and lists. The context of the programming task was to move a soccer ball object on a computed trajectory path and check whether a fixed, marked point in the goal was hit with the trajectory. The trajectory is represented in a list variable with 36 elements, denoting the flight heights (coordinate on the y-axis) that are given in 10-point distances (coordinate change on the x-axis). The computation of the trajectory, based on user input values, was fully prepared. The coordinates of the fixed, marked point were given. The students received an information sheet that specified the programming task, received a prepared `.sb3` Scratch file to open in the instrumented Scratch 3 environment, and had a maximum of 20 minutes to complete the task.

[1]Link to the publicly-available measurement server: http://seqtrex.aau.at/

Table 1: Cohorts of students participating in the trial.

Cohort	*Age*	*N*	*Solution None*	*Solution Linear*	*Solution Loop*
Interns 2018	15-18	8	1	0	7
Interns 2019	15-18	6	3	1	2
School Upper	14-15	8	6	1	1
School Lower	13-14	20	14	6	0
Total		42	24	8	10

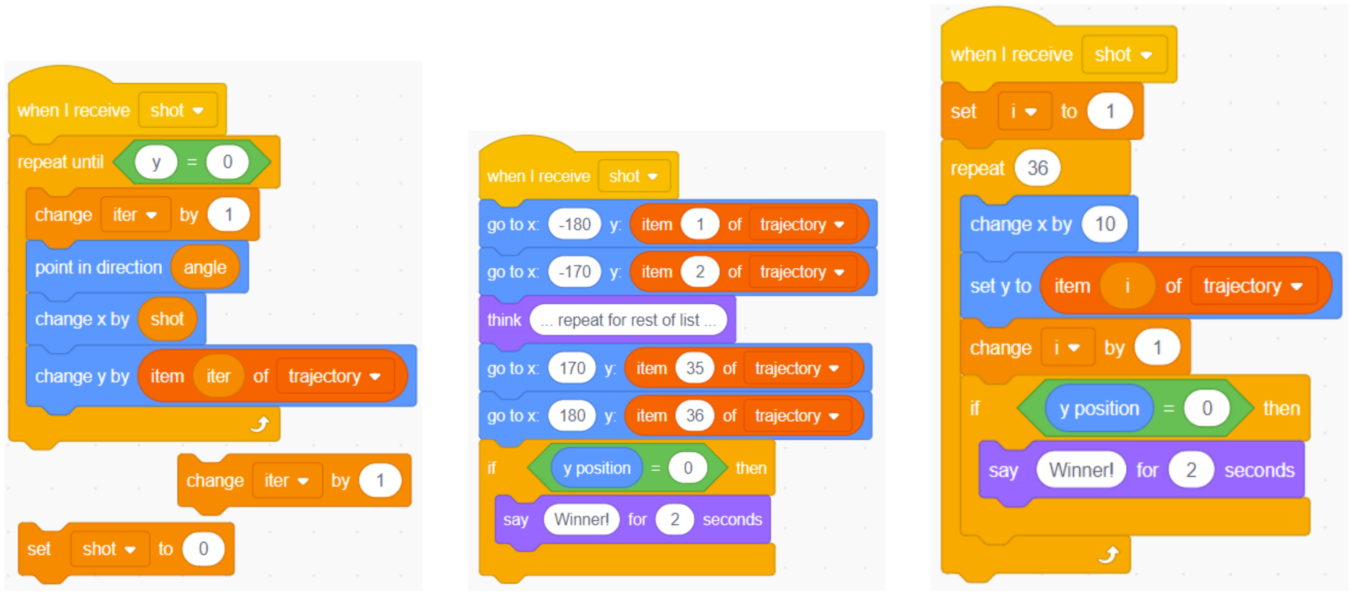
Table 1 shows the student cohorts participating in the programming trial. The cohort *Interns 2018* was collected in our previous work [21] and has been replayed in the proposed Scratch 3 measurement framework to obtain comparable data.

The first two cohorts of Table 1 consist of upper secondary school students that worked as interns in the respective year at our department. The students had varied programming skills, ranging from no programming skills to programming skills in multiple text-based programming languages, and were measured individually.

The second two cohorts of Table 1 consist of lower and upper secondary school students, measured in a computer science class setting in their respective school. In both classes, block-based programming in Scratch had previously been covered by instructions. The upper secondary school students (cohort *School Upper*) were measured during a compulsory computer science class. The lower secondary school students (cohort *School Lower*) were measured during an elective computer science class.

Figure 1 shows three exemplary student-developed programs, one of each group. The programs were extracted after the recorded last program change. The programs of students that did not solve the programming task are very diverse. They could feature incorrectly applied programming concepts, correctly applied programming concepts with a wrong understanding of the problem (as the one in Figure 1 (a)), unsuccessful attempts to solve the problem in a linear fashion (mostly not taking into consideration the specified coordinate changes), and a seemingly random sequence of blocks that do not contribute to a solution of the problem.

Linear solutions to the problem have been implemented in two primary ways. The first one, similar to Figure 1 (b), uses a sequence of *go to* blocks and changes x- and y-coordinates at the same time. This way, x-coordinates have to be hard-coded in the move blocks, although the task specifies constant change of the x-coordinate by 10. The second one uses pairs of move blocks instead of the



```
when I receive shot
repeat until y = 0
    change iter by 1
    point in direction angle
    change x by shot
    change y by item iter of trajectory

change iter by 1

set shot to 0
```

(a) No solution

```
when I receive shot
go to x: -180 y: item 1 of trajectory
go to x: -170 y: item 2 of trajectory
think ... repeat for rest of list ...
go to x: 170 y: item 35 of trajectory
go to x: 180 y: item 36 of trajectory
if y position = 0 then
    say Winner! for 2 seconds
```

(b) Exemplary linear solution

```
when I receive shot
set i to 1
repeat 36
    change x by 10
    set y to item i of trajectory
    change i by 1
    if y position = 0 then
        say Winner! for 2 seconds
```

(c) Exemplary loop solution

Figure 1: Three exemplary last program states of students participating in the trial. The left program contains no solution, although programming concepts like loops and variables are used. The middle program shows a snippet of a linear solution. The right program shows a looping solution, correctly applying the programming concepts of loops, conditionals and variables.

combined *go to* block, similar to a part of the loop solution Figure 1 (c). This way, x-coordinates can be changed by 10 units each, while the y-coordinate can be set to the next element of the list.

All loop solutions are very similar to the one shown in Figure 1 (c). Compared to the linear solution, the loop solution is distinguished by the usage of a loop (with fixed size as specified) and an iteration variable to program the movement along the trajectory.

3.3 Categorization of Recorded Data

The raw data recorded during the programming trial consists of mouse and keyboard interactions since the last change, the current program change and a textual representation of the whole program. Each program change is a **(E) block listen event**. A meaningful sequence of such block listen events is categorized into **(P) program change events**, as described below. Table 2 shows the three categories to classify program changes.

For each program change, the block types of interacted program blocks are categorized. Most **(T) block types** correspond to the Scratch 3 block type categories: **T1** *motion*, **T2** *looks*, **T6** *event*, **T10** *operator*, **T11** *sound*, **T12** *sensing*, **T13** *user-defined*, **T14** *extensions*. However, some block types are more fine-grained. The Scratch 3 category *Variables* (internally called data) is split into two categories: **T4** *var* for blocks that deal with variables, and **T5** *lists* for blocks that deal with lists. The Scratch 3 category *Control* is split into three categories: **T8** *loop* for loop blocks, **T9** *conditional* for conditional branching blocks, and **T7** *control* for all other blocks of this Scratch 3 category. The block type **T3** *pen* is a Scratch 3 extension, but is included as a separate block type to ensure backwards comparability with our previous study [21].

Most of the **(E) block listen events** are directly derived from the block change events observed by the Scratch virtual machine [29] when a user interacts with the programming environment and makes changes to the program blocks.

E1 *create* events capture block creations, **E2** *change* capture changes to block fields, **E3** *move* capture block movements after block drags, **E4** *delete* capture block deletions, **E5** *end-drag* capture block drags. **E6** *outside-drag* and **E7** *end-drag-onto* capture block drags that leave the program area and can result in copying blocks to other objects. The events **E8–E12** handle variable and list management, the events **E13–E16** handle comment management.

Interactions that start and stop the program execution are additionally instrumented and captured. **E17** *stackclick* events represent a user's click on a stack of blocks and executes or evaluates them, **E18** *greenflag* events represent a user's click on the greenflag symbol that starts general execution, and **E19** *stopall* events represent a user's click on the stopall symbol that stops general execution.

The block listen events **E1–E5** represent single actions and have to be bundled to form meaningful **(P) program change events**. Most of the program change events (**P1–P10**) are divided in two modes. Change events with suffix *-program* indicate that the change affects blocks connected to a main executable block (so-called *hat* blocks[2]). Change events with suffix *-nonprogram* indicate that the change affects blocks in the non-executable part of the program.

The addition of new blocks to the program (**P1**, **P2**) is classified from the following sequence: **E1** the creation of the new block from the Scratch 3 category, **E5** the end of the drag event and **E3** the movement of the block to the new destination.

[2]Scratch wiki regarding hat blocks: https://en.scratch-wiki.info/wiki/Hat_Block

Table 2: Overview of categories for classifying programming interactions.

(T) Block Type

T1 motion	**T4** var	**T7** control	**T10** operator	**T13** user-defined
T2 looks	**T5** lists	**T8** loop	**T11** sound	**T14** extensions
T3 pen	**T6** event	**T9** conditional	**T12** sensing	

(E) Block Listen Events

E1 create	**E5** end-drag	**E9** create-var-global	**E13** create-comment	**E17** stackclick
E2 change	**E6** outside-drag	**E10** rename-var-local	**E14** change-comment	**E18** greenflag
E3 move	**E7** end-drag-onto	**E11** rename-var-global	**E15** move-comment	**E19** stopall
E4 delete	**E8** create-var-local	**E12** delete-var	**E16** delete-comment	

(P) Program Change Events

P1 add-program	**P4** attach-nonprogram	**P7** reorder-program	**P10** delete-nonprogram	**P13** block-move
P2 add-nonprogram	**P5** detach-program	**P8** reorder-nonprogram	**P11** block-change	**P14** block-click
P3 attach-program	**P6** detach-nonprogram	**P9** delete-program	**P12** immediate-delete	

The block listen event **E3** *move* specifically spawns a number of program change events, depending on the context of the block movement. Attaching a block to another block (**P3**, **P4**) is classified from a move event that adds a new parent block to the moved block and adds a new next block to the block attached to. Detaching a block from another block (**P5**, **P6**) is classified from a move event that removes the parent block from the moved block and removes the next block from the block detached from. Reordering of a block stack (**P7**, **P8**) is classified from a move event that specifies old and new parent blocks in the moved block. Attach and reordering are often preceded by detach events, when blocks are moved from one program part to another. A simple block move (**P13**) is classified from a move event that does not change block order.

The deletion of blocks (**P9**, **P10**) is classified from the following sequence: **E5** the end of the drag event, **E3** the movement of the block to the new destination and **E4** deletion of the block.

Detailed changes to block field parameters (**P11**) are classified from **E2** change events. Program change events spawned from **E3** *move* can also change block fields (for example moving a variable block into a block field), but are then classified as the respective move event (attach, detach or reorder).

Table 3: Simplified example categorization (block listen events not shown) for creation and attachment of a *go to* block with access to list element 3 (see also Figure 1 (b)).

User Action	*Categorized Sequence*
Create *go to* block	**P1** *add-program*, **T1** *motion*
Attach it after block	**P3** *attach-program*, **T1** *motion* (parent)
	P3 *attach-program*, **T1** *motion* (next)
Edit *x field* (−160)	**P11** *block-change*, **T1** *motion*
Create *list access* block	**P1** *add-program*, **T5** *lists*
Attach it in *y field*	**P3** *attach-program*, **T5** *lists*
	P3 *attach-program*, **T1** *motion*
Edit *list access field* (3)	**P11** *block-change*, **T5** *lists*

Immediate deletion of blocks (**P12**) is the creation and deletion of blocks from the Scratch 3 block categories with the same drag event, and is similar to block deletion with a preceding create event. It is classified from the following sequence: **E1** the creation of the new block, **E5** the end of the drag event, **E3** the movement of the block to the new destination and **E4** deletion of the block. The difference to normal deletion events (**P9**, **P10**) is that immediate deletion does not cause any program block changes.

Block clicks that execute the stack of blocks (**P14**) are classified from **E17** *stackclick* and represent the same event.

The analysis data for each category of Table 2 consists of the fractions of category types, classified from each student's sequence of program changes. As in our previous work [21], we additionally computed the maximum number of used block types, and the geometric mean change rates of used block types in the executable and non-executable program.

Table 3 shows a simplified categorization (only program change events aggregated from block listen events are shown) of an example creation and attachment of another *go to* block with access to list element 3, comparable to those in Figure 1 (b).

3.4 Fine-grained Sequence Data

Besides the fractions of category types, the other type of data used for analysis is a more fine-grained sequence of program changes obtained from a combination of **(P) program change events** and **(T) block types**. The events to add blocks (**P1**, **P2**), to delete blocks (**P9**, **P10**, **P12**), and to click blocks (**14**) are simply enriched with the block type. Block change events (**P11**) are enriched with the block type and the type of change (*field* for editable block field parameters, *input* block fields with drop-down input parameters). The events to attach (**P3**, **P4**), detach (**P5**, **P6**) and reorder (**P7**, **P8**) blocks are enriched with block types for source and destination of the movement, and type of change (*order* for block order changes, *input* for block field parameter changes). Block move events (**P13**) are not considered for this type of data as they do not change the structure of the program.

Analysis data from these fine-grained sequences consists of occurrences of frequent patterns that occur in the sequences of two or more students. They are stored as a vector of occurrence (1 for occurring pattern, 0 for non-occurring pattern).

4 TRIAL RESULTS

In this section we present the results of the trial in a descriptive way. We first present quantitative results (correlation and cluster analysis), and then present patterns in the students' fine-grained sequence data.

4.1 Quantitative Results

The raw data after classification consists of a mean number of 326 ± 194 **(E) block listen events**, and a mean number of 299 ± 197 **(P) program change events** per student.

Before performing data analysis, we selected a subset of **(E) block listen events** metrics to be included in the analysis: **E8** and **E9** (variable creation), **E17–E19** (program execution). The other metrics of this category are either captured in **(P)** program change events (**E1–E5**) or do not occur in the data set.

We tested normality of the data (fractions of category types) with the Shapiro-Wilk test and rejected the null hypothesis of normality for most of the metrics at $p < 0.05$. Because of this, we performed correlation analysis with Spearman's rank correlation and a significance of $p < 0.05$. In order to control the false discovery rate (as is called for with pair-wise tests of 34 metrics), we used the Benjamini-Hochberg procedure [5]. A power test shows that correlations of 0.5164 or greater have sufficient power (with a significance of 0.05, test power of 0.95 and 42 subjects [11]).

We first investigated the correlations between the students' success (numerical encoding: 0 for no solution, 0.5 for linear solution, 1 for loop solution) and the fractions of category types to see whether particular high or low frequencies might indicate successful students or such in need for support. Table 5 presents the correlations.

There are moderate positive correlations between **Success** and the block types needed to solve the trial (**T2** looks, **T4** var, **T8** loop, **T9** conditional, and **T10** operator blocks), and one moderate negative correlation with **T6** event blocks. This shows that successful students interact with relevant block types more often, but do not interact with event blocks as often.

The strong positive correlation between **Success** and **E9** global variable creation again discriminates successful students, who created variables more often, based on a programming concept needed to solve the trial. The moderate negative correlation to **E18** greenflag execution shows that successful students use the greenflag execution not as often.

Regarding program change events, only **P7** program reordering shows a moderate positive correlation to the students' success.

A moderate positive correlation shows that successful students tend to use more block types in their program (**MaxBlockTypes**).

All but the fractions of **E18** greenflag execution and **T6** events block usage are related measures, exhibiting moderate to strong positive correlations between each other.

Next we performed cluster analysis with the students' programming sequence data, using k-means. Notably, we did not include the measure of success in the cluster analysis. Using the elbow technique, 4 was found to be the appropriate number of clusters for the data [16]. Table 4 presents the results of the cluster analysis.

Table 4: Results of clustering students' programming sequence data with k-means, $n = 42$. Cluster 2 represents the linear solution, cluster 3 represents the loop solution. Clusters 1 and 4 capture students who did not solve the trial.

Cluster	*Solution None*	*Solution Linear*	*Solution Loop*	*Total*
1	14	1		15
2	4	7		11
3	1		10	11
4	5			5
Total	24	8	10	42

There is one cluster for each group of students that successfully solved the trial (cluster 2 represents the linear solution, cluster 3 represents the loop solution). Only a small number of students who did not solve the trial are also grouped in those clusters (five students in total). The other two clusters (cluster 1 and cluster 4) mainly capture two groups of students who did not solve the trial.

We evaluated the cluster groups by comparing box plots of the metrics used for clustering, and tested for significant differences with the Mann-Whitney-U test and a significance of $p < 0.05$. Figure 2 shows boxplots that contain at least one significant difference.

We first describe the clusters of students who did not solve the trial. Both clusters use **T1** motion blocks significantly more often than cluster 3. Cluster 1 shows some use of **T4** variable, **T5** lists, **T8** loop and **T10** operator blocks, cluster 4 shows very small usage of those types. Both clusters have the highest fraction of execution interactions: Cluster 1 performs **E17** stackclicks, cluster 4 performs **E18** greenflag executions with a significant difference compared to clusters 1 and 3. Compared to cluster 1, cluster 4 students detach blocks from the non-executable program (**P6**) less often, but have a significantly higher fraction of block deletions from the non-executable program (**P10**). Cluster 1 has a higher maximum number of block types in the executable program and a higher geometric mean type change rate compared to cluster 4.

We next describe cluster 2 of students with a linear solution. Cluster 2 uses significantly more **T1** motion blocks compared to clusters 1 and 3, significantly more **T5** lists blocks compared to cluster 1, and significantly less **T6** event blocks compared to clusters 1 and 4. Cluster 2 has a significantly higher fraction of **P1** changes that add blocks to the executable program, compared to cluster 1. Cluster 2 also has significantly lower fractions of **P6** detaching blocks from the non-executable program and of **P13** block move events, compared to cluster 3. Cluster 2 has a significantly smaller geometric mean type change rate compared to cluster 1.

Lastly, we describe cluster 3 of students with a loop solution. In line with the correlations for success, cluster 3 has the highest usage fraction of **T2** looks, **T4** var, **T8** loop, **T9** conditional and **T10** operator blocks, the highest fraction of **E9** global variable creation and the highest maximum number of block types in the executable program. Cluster 3 has a significantly higher geometric mean type change rate than any other cluster.

Table 5: All significant Spearman correlations ($p < 0.05$, corrected with the Benjamini-Hochberg procedure to control the false discovery rate) between fractions of category types and success in the trial.

n=42	***Success***	***T2***	***T4***	***T6***	***T8***	***T9***	***T10***	***E9***	***E18***	***P7***	***MaxBlockTypes***
Success	—	**0.55**	**0.44**	**-0.40**	**0.43**	**0.58**	**0.47**	**0.72**	**-0.43**	**0.48**	**0.55**
T2 looks	**0.55**	—	**0.49**		**0.60**	**0.84**	**0.68**	**0.62**		**0.55**	**0.72**
T4 var	**0.44**	**0.49**	—		**0.54**	**0.56**	**0.56**	**0.66**		**0.44**	**0.73**
T6 event	**-0.40**			—							
T8 loop	**0.43**	**0.60**	**0.54**		—	**0.59**	**0.57**	**0.66**		**0.59**	**0.68**
T9 conditional	**0.58**	**0.84**	**0.56**		**0.59**	—	**0.82**	**0.63**		**0.48**	**0.78**
T10 operator	**0.47**	**0.68**	**0.56**		**0.57**	**0.82**	—	**0.57**		**0.51**	**0.81**
E9 create-var-global	**0.72**	**0.62**	**0.66**		**0.66**	**0.63**	**0.57**	—		**0.50**	**0.71**
E18 greenflag	**-0.43**								—		
P7 reorder-program	**0.48**	**0.55**	**0.44**		**0.59**	**0.48**	**0.51**	**0.50**		—	**0.50**
MaxBlockTypes	**0.55**	**0.72**	**0.73**		**0.68**	**0.78**	**0.81**	**0.71**		**0.50**	—

4.2 Recurring Programming Sequences

The fine-grained sequence data described in Section 3.4 yields 354 different program change types that occur in the students' programming sequences. A total of 6727 frequent patterns have been found in the students' programming sequences, with a length of 2 to 51 program changes. Our goal is to identify patterns in the programming sequences of successful students, especially of those using a loop solution to solve the trial (cluster 3).

After a manual investigation of the frequent patterns, we saw that patterns of length 2 or 3 were too small to form programming patterns and filtered them out (1211 patterns). We then realized that only trivial patterns (like repeated change of a block parameter field during typing) were recurring for a majority of successful students. Actually, only 20 of the remaining 5516 patterns (0.36%) are used by half or more of the students constructing a loop solution.

We still set out to find patterns relevant to a successful trial completion with loop solution, and filtered the patterns, only retaining those that include at least one block of type **T4** var (88 patterns), **T8** loop (12 patterns), **T9** conditional (90 patterns) or **T10** operator (43 patterns). We manually investigated those filtered patterns and identified 10 small-scale patterns with a length of 4 to 14 which capture important steps in the construction of the loop solution.

5 patterns are exclusive to constructing the loop solution: adding a loop and editing the iteration number (2 patterns with 2 students each), attaching conditional and operator blocks after the loop to solve task 2 (3 students), adding list and variable blocks to access the list with a variable (2 students), adding variable changes and conditional blocks to solve task 2 (2 students).

2 patterns include students constructing the loop as well as the linear solution: adding a list block to a motion block field (1 student of each solution type), adding the motion variable *y position* to an operator block and appending output for task 2 of the trial (two students of loop solution and one of linear solution).

3 patterns include students constructing the loop solution and students not solving the trial. The students use conditional blocks to solve task 2 of the trial (2, 2, and 1 student/s of cluster 1; 6, 3, and 2 students of cluster 3 respectively).

5 DISCUSSION OF FINDINGS

After an initial description of the trial results in Section 4, we now continue with a discussion of the findings and focus on answering the research questions for this paper.

5.1 Identification of Struggling Students

The first research question is: *How can block-based programming sequence data from an IDE-based learning analytics setting be used to identify students in need of support?*

To resolve this research question, we give a task-based summary of clusters 1 and 4 and their differences, using Figure 2. We put the summary into the frame of neo-Piagetian stages of development following Lister [24].

Students less successful in the trial execute the program more often. Cluster 1 (4% greenflag, 5% stackclick actions) and cluster 4 (9% greenflag, 2% stackclick actions) both execute the program with about 10% of all actions. Repeated, frequent execution of the program could hint at the *pre-tracing* stage [24], when novice programmers have no consistent understanding of basic programming concepts and the program execution. Executing the program could be a way to test hypotheses regarding program changes.

The usage fractions of block types needed to solve the trial (**T4** var, **T5** lists, **T8** loop, **T9** conditional, and **T10** operator blocks) discriminate between students solving the trial (cluster 3 has a mean usage fraction of 10% per type) and students not solving the trial. The latter can be further divided into students of cluster 1 with a mean usage fraction of 5% per type, and students of cluster 4 with no usage of these block types.

Summarizing, cluster 4 encompasses novice programmers that struggle with the application of basic programming concepts and with consistent reasoning about their program code. These students use the block types **T1** motion, **T3** pen, and **T6** events nearly exclusively, and execute their program frequently with greenflag execution (main program execution). They need general support to increase their programming skills and understanding of basic programming concepts. Students in this cluster could be described with the *pre-tracing* stage of neo-Piagetian development [24].

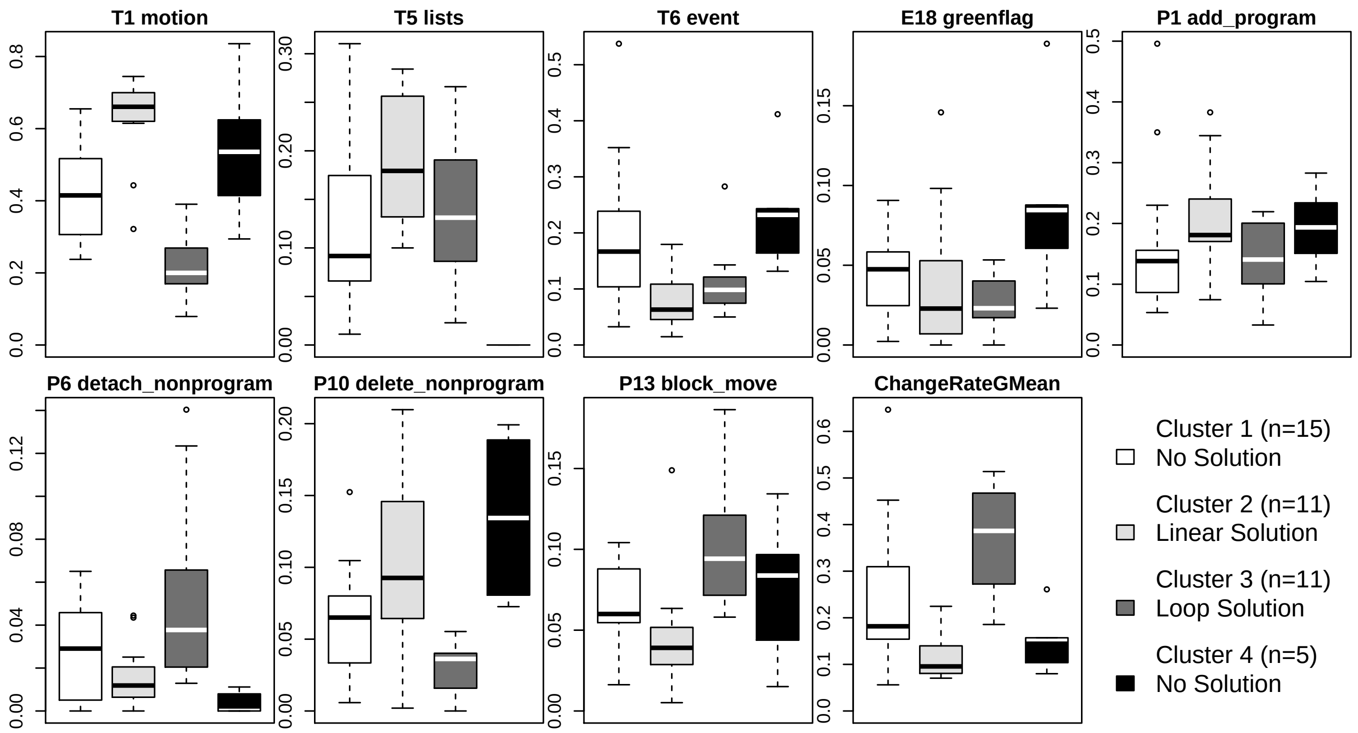


Figure 2: Distribution of students' usage fractions for selected category types, divided into the four clusters. All presented category types exhibit at least one significant difference between any of the clusters.

Cluster 1 on the other hand encompasses novice programmers that are more familiar with basic programming concepts and use them, but cannot apply them consistently. These students execute their program frequently by clicking on blocks. They need more specific support, depending on the programming concepts they are struggling with. An indication is a particularly high or low usage fraction for specific block types like variables or loops. Students in this cluster could be on their path between the stages of *pre-tracing* and *tracing* [24], as different neo-Piagetian stages of development can be overlapping [36]. They already apply programming concepts but are not capable of correct and consistent code tracing yet.

Answering the first research question, the following metrics identify students in need of support:

- Repeated, frequent execution of the program. In the trial, frequent execution was measured at about 10% of all actions. This could signal that students struggle with code tracing and reasoning, and with the application of basic concepts.
- Limited usage of different block types that represent programming concepts (like variables, lists, loops, conditional branching, operators). In the trial, limited usage was measured at less than 5% of all block actions. This could indicate a lack of understanding of those concepts.

5.2 Discrimination of Successful Students

The second research question is: *What programming sequence metrics and recurring patterns discriminate students successful in the trial?*

As we have shown in Section 4.2, no recurring patterns could be identified that discriminate a majority of successful students. But we use the programming sequence metrics to resolve this research question. We give a task-based summary of clusters 2 and 3 and their differences, using Figure 2. We put the summary into the frame of neo-Piagetian stages of development following Lister [24].

Students more successful in the trial execute the program less often. Cluster 2 (4% greenflag, 1% stackclick actions) and cluster 3 (3% greenflag, 3% stackclick actions) have the lowest mean values of program execution actions. Interpreted with neo-Piagetian theory, those students do not rely on program execution to understand their program. This shows the students' ability to trace and reason about their code, attributing at least the *tracing* stage to them [24].

Cluster 3 has a balanced usage fraction of block types needed to solve the trial (**T4** var, **T5** lists, **T8** loop, **T9** conditional, and **T10** operator blocks) with a mean 10% per type. This is also reflected in a high geometric mean type change rate. In contrast, the construction of the linear solution of cluster 2 is characterized by higher usage fractions of list blocks (19%) and motion blocks (62%).

Summarizing, cluster 2 encompasses novice programmers who can correctly apply some basic programming concepts and can consistently trace and reason about their code. They solved the trial by combining **T1** motion and **T5** lists blocks in a linear fashion without any loop. Students in this cluster have developed some skills of the *tracing* stage. They need specific support in programming concepts, especially with variables and loops, as they are likely still in the *pre-tracing* stage [24] with regard to these concepts.

Cluster 3 encompasses experienced programmers who can apply all basic programming concepts and can consistently trace and reason about their code. They solved the trial with seven to eight different block types, and have a significantly higher geometric mean type change rate than any other cluster. This is in line with our previous findings [21] and further confirms that the proposed type change rate metric is a discriminator of students with high programming skills, when using this programming task. Students in this cluster have likely reached the *tracing* stage, and are on their path to develop *post-tracing* skills like deductive reasoning [24].

Answering the second research question, the following metrics discriminate successful students:

- Sparing execution of the program. In the trial, successful students executed the program with less than 6% of all actions. This indicates that better students do not need to execute the program in order to reason about and understand it.
- High geometric mean type change rate, which is a higher fraction of changes that alter used block types. A mean value of 37% was measured for students of cluster 3 in the trial. This indicates that students who use different block types in a balanced way are more likely to solve similar tasks.

5.3 Implications for Instructors and IDE

The aim of our research is to make individual support for students during programming possible. The results of this work are a first step towards this goal. We computed metrics from students' programming sequences and identify struggling students based on their high frequency of execution and limited usage of different block types. We envision two implications from our findings.

The first is an *educator dashboard* that presents programming sequence metrics to instructors during the students' task completion. Instructors can get alerted on high frequencies of execution or repeated execution, and offer support. Potential support for students of clusters 1 and 4 includes guided step-by-step execution to improve their code tracing skills, or instructions how to use specific programming concepts. Filvà et al. [12] also use dashboard reporting, but our approach can be employed during task completion.

The second is an IDE designed for learning assignments. Instructors could specify sample solutions for an assignment, and the IDE could monitor students' programming. Whenever an assignment requires specific block types (like loops or lists) or block interactions (like operators attached to conditional blocks) to be solved and the IDE detects an underrepresented usage of corresponding blocks, it could notify the student or display helping documentation.

5.4 Threats to Validity

Here we describe external and internal threats to the validity of the described results. The first threat is concerned with the group of experimental subjects. The cohorts are diverse and vary in age, level of programming skills, experience with Scratch 3 and experimental context (internship and class setting). This uncontrolled variance is a threat to the generalizability of the results, especially regarding the linear solution (the majority of students belong to the same class). Regarding the loop solution, the threat is reduced as the students are drawn from different educational backgrounds (vocational schools, grammar schools) and still produce a comparable solution.

The second threat is concerned with the programming task. The task was specifically crafted to require multiple block types and different programming concepts to be solved. Our results may not be generalizable to arbitrary programming problems. We use a fixed programming task to cover the use of variables, lists, loops, conditionals and operators with a common functional aim. This makes context-sensitive analysis possible, but prevents us to find general programming strategies and discriminating metrics.

Regarding internal validity, the statistical analysis was conducted with the software R, and we report results with a significance of $p < 0.05$. After rejecting normality for the data, we only applied non-parametric statistics. We used the Benjamini-Hochberg procedure to control the false discovery rate [5]. As not all correlations have sufficient power (0.5164 with a significance of 0.05, a test power of 0.95, and 42 subjects [11]), we also performed cluster analysis and evaluated cluster differences with Mann-Whitney-U tests.

The patterns found in the fine-grained sequence data were analyzed manually by the authors. The resulting data was not suitable to answer the research questions of this paper. A major point is the noise in the fine-grained data. Additional data preparation and data analysis with this in mind is subject to future work.

6 CONCLUSION

We measured four cohorts of lower and upper secondary school students solving a programming trial in Scratch 3 and recorded their programming sequences with an IDE-based learning analytics framework. Our goals are to identify students in need of support students, and to discriminate successful students based on programming sequence metrics and recurring patterns. The advantage of our learning analytics approach is the possibility to perform an evaluation *during* and not only after task completion.

We categorized the programming sequence data based on used block types and types of program change, and used the resulting fractions of category types for rank correlation analysis (Table 5) with the students' success. We found moderate positive correlations to block types that represent programming concepts (variable, loop, conditional, and operator), correlations to specific program change actions (strong positive correlation with variable creation, moderate negative correlation with main program execution) and a moderate positive correlation to the maximum number of used block types.

We also employed clustering with k-means, obtaining 4 clusters (Table 4) of students, representing different trial solutions. Two clusters describe students not successful in the programming trial. Cluster 4 ($n = 5$) is composed of students at the *pre-tracing* stage: they use no block types representing programming concepts, and struggle the most with the programming trial. They benefit from general support regarding basic programming concepts. Cluster 1 ($n = 15$) is composed of students likely between the stages of *pre-tracing* and *tracing*: they use block types representing programming concepts, but cannot apply them consistently. They benefit from individual support regarding programming concepts. Both clusters feature a repeated, frequent execution of the program with about 10% of all actions, and use block types that represent programming concepts in less than 5% of all block actions.

The two other clusters describe students successful in the programming trial. Cluster 2 ($n = 11$) captures students programming a linear solution. They are at the stage of *tracing* for sequential programming, but are likely at the stage of *pre-tracing* for other programming concepts and need specific support. Cluster 3 ($n = 11$) is composed of students programming a loop solution. They can consistently apply all basic programming concepts and have likely reached the *tracing* stage. They feature a balanced use of all block types needed to solve the trial (about 10% per block type), and have a high rate of changes in the used block types (about 37%). Both clusters execute the program with less than 6% of all actions.

Following these results, the proposed IDE-based learning analytics framework can be used to individually support students during task completion based on their programming sequence. Repeated, frequent execution of the program or limited usage of block types that are necessary to solve a given task can indicate a student's need for support. This can trigger the instructor or even the IDE to provide more specific instructions or additional documentation.

To discriminate students constructing the loop solution, we also investigated recurring patterns in fine-grained programming sequences (types of program changes enriched with interacted block types). After excluding small and trivial patterns, only 20 of the remaining 5516 patterns (0.36%) are used by half or more of the students with a loop solution. After only considering patterns that use block types relevant to the loop solution, we found 10 small-scale patterns with a length of 4 to 14 program changes which describe important steps in the construction of the loop solution. This result shows that small-scale programming strategies are very diverse.

Going forward, we intend to conduct additional programming trials with a diverse set of programming tasks to evaluate the generalizability of the described measurement approach. We plan to conduct qualitative studies with think-aloud programming and post-programming interviews to explain emerging programming strategies from the student's view. We envision an evaluation dashboard for educators to support students during task completion.

ACKNOWLEDGMENTS

We want to express our gratitude to the participating students and teachers for the possibility to conduct the programming trial.

REFERENCES

[1] Efthimia Aivaloglou and Felienne Hermans. 2016. How Kids Code and How We Know. *Proceedings ICER '16* (2016), 53–61.
[2] L. W. Anderson, D. R. Krathwohl, P. W. Airasian, K. A. Cruikshank, R. E. Mayer, Pintrich P. R., J. D. Raths, and M. C. Wittrock. 2001. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives* (1 ed.). New York: Longman.
[3] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. 2015. From Scratch to "real" programming. *ACM TOCE* 14, 4 (2015), 1–15.
[4] R. Baker and K. Yacef. 2009. The State of Educational Data Mining in 2009: A review and future visions. *JEDM* 1, 1 (2009), 3–17.
[5] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300.
[6] J. Biggs and K. Collis. 1982. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)* (1 ed.). NY Academic Press.
[7] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. 2014. Programming Pluralism: Using Learning Analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences* 23, 4 (2014), 561–599.
[8] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. 2013. Hairball: Lint-inspired Static Analysis of Scratch Projects. *Proceeding SIGCSE '13* (2013), 215.
[9] Andreas Bollin, Heike Demarle-Meusel, Max Kesselbacher, Corinna Mößlacher, Marianne Rohrer, and Julia Sylle. 2018. The bebras contest in Austria - Do personality, self-concept and general interests play an influential role?. In *The 11th Conference on Informatics in Schools - ISSEP '18*, Vol. 11169 LNCS. 283–294.
[10] Code.Org. 2019. code.org. Retrieved June 2019 from https://code.org/
[11] Jacob Cohen. 1977. Differences between Correlation Coefficients. In *Statistical Power Analysis for the Behav. Sciences*, Jacob Cohen (Ed.). Acad. Press, 109 – 143.
[12] Daniel A. Filvà, Marc A. Forment, Francisco J. García-Peñalvo, David F. Escudero, and María J. Casañ. 2019. Clickstream for learning analytics to assess students' behavior with Scratch. *Future Generation Computer Systems* 93 (2019), 673–686.
[13] Google Developers. 2019. Blockly. Retrieved June 2019 from https://developers.google.com/blockly/
[14] Andreas Grillenberger and Ralf Romeike. 2017. Real-Time Data Analyses in Secondary Schools Using a Block-Based Programming Language. In *Informatics in Schools: Focus on Learning Programming*, Valentina Dagienė and Arto Hellas (Eds.). Springer International Publishing, Cham, 207–218.
[15] Shuchi Grover and Satabdi Basu. 2017. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. *ACM SIGCSE Technical Symposium on Computer Science Education 2017* (2017), 267–272.
[16] Jiawei Han, Micheline Kamber, and Jian Pei. 2011. *Data Mining: Concepts and Techniques* (3 ed.). Morgan Kaufmann.
[17] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. *IEEE ICPC* July (2016), 1–10.
[18] Felienne Hermans, Kathryn T. Stolee, and David Hoepelman. 2016. Smells in block-based programming languages. *Proceedings of IEEE Symposium VL/HCC* November (2016), 68–72.
[19] C. D. Hundhausen, D. M. Olivares, and A. S. Carter. 2017. IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda. *ACM Trans. Comput. Educ* 17, 26 (2017), 1–26.
[20] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. *ITiCSE WGR'16* (2015), 41–63.
[21] Max Kesselbacher and Andreas Bollin. 2019. Quantifying Patterns and Programming Strategies in Block-based Programming Environments. In *Companion Proceedings ICSE '19*. IEEE Press, Piscataway, NJ, USA, 254–255.
[22] K. Kontogiannis. 1997. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the Fourth Working Conference on Reverse Engineering*. IEEE Computer Society, Washington, DC, USA, 44–54.
[23] Raymond Lister. 2011. Concrete and other neo-piagetian forms of reasoning in the novice programmer. *Conferences in Research and Practice in Information Technology Series* 114, January 2011 (2011), 9–18.
[24] Raymond Lister. 2016. Toward a developmental epistemology of computer programming. *WiPSCE '16* (2016), 5–16.
[25] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of programming in scratch. *ITiCSE '11* (2011), 168.
[26] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2013. Learning computer science concepts with Scratch. *Computer Science Education* 23, 3 (2013), 239–264.
[27] MIT Media Lab. 2019. Scratch. Retrieved June 2019 from https://scratch.mit.edu/
[28] MIT Media Lab. 2019. Scratch-Blocks. Retrieved June 2019 from https://github.com/LLK/scratch-blocks
[29] MIT Media Lab. 2019. Scratch-VM. Retrieved June 2019 from https://github.com/LLK/scratch-vm
[30] J. Moreno and G. Robles. 2014. Automatic detection of bad programming habits in scratch: A preliminary study. In *2014 IEEE FIE Conference Proceedings*. 1–4.
[31] S. Papavlasopoulou, M. N. Giannakos, and L. Jaccheri. 2018. Discovering children's competences in coding through the analysis of Scratch projects. In *2018 IEEE Global Engineering Education Conference (EDUCON)*. 1127–1133.
[32] Simon Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, USA.
[33] Viera K. Proulx. 2000. Programming patterns and design patterns in the introductory computer science course. *SIGCSE Bull.* 32, 1 (March 2000), 80–84.
[34] Kelly Rivers, Erik Harpstead, and Ken Koedinger. 2016. Learning Curve Analysis for Programming. In *Proceedings ICER '16*. 143–151.
[35] Alaaeddin Swidan, Alexander Serebrenik, and Felienne Hermans. 2017. How do Scratch programmers name variables and procedures? *IEEE SCAM 2017* October (2017), 51–60.
[36] Donna Teague and Raymond Lister. 2014. Longitudinal think aloud study of a novice programmer. *CRPIT* 148 (2014), 41–50.
[37] Giovanni M. Troiano, Sam Snodgrass, Erinc Argimak, Gregorio Robles, Gillian Smith, Michael Cassidy, Eli Tucker-Raymond, Gillian Puttick, and Casper Harteveld. 2019. Is My Game OK Dr. Scratch?: Exploring Programming and Computational Thinking Development via Metrics in Student-Designed Serious Games for STEM. In *Proceedings IDC '19*. ACM, New York, NY, USA, 208–219.
[38] Christopher Watson and Frederick W.B. Li. 2014. Failure rates in introductory programming revisited. In *ITiCSE '14*. ACM, New York, USA, 39–44.