Common Patterns in Block-Based Robot Programs

Florian Obermüller
University of Passau
Passau, Germany

Robert Pernerstorfer
University of Passau
Passau, Germany

Lisa Bailey
University of Passau
Passau, Germany

Ute Heuer
University of Passau
Passau, Germany

Gordon Fraser
University of Passau
Passau, Germany

ABSTRACT

Programmable robots are engaging and fun to play with, interact with the real world, and are therefore well suited to introduce young learners to programming. Introductory robot programming languages often extend existing block-based languages such as Scratch. While teaching programming with such languages is well established, the interaction with the real world in robot programs leads to specific challenges, for which learners and educators may require assistance and feedback. A practical approach to provide this feedback is by identifying and pointing out patterns in the code that are indicative of good or bad solutions. While such patterns have been defined for regular block-based programs, robot-specific programming aspects have not been considered so far. The aim of this paper is therefore to identify patterns specific to robot programming for the Scratch-based mBlock programming language, which is used for the popular mBot and Codey Rocky robots. We identify: (1) 26 *bug patterns*, which indicate erroneous code; (2) three *code smells*, which indicate code that may work but is written in a confusing or difficult to understand way; and (3) 18 *code perfumes*, which indicate aspects of code that are likely good. We extend the LitterBox analysis framework to automatically identify these patterns in mBlock programs. Evaluated on a dataset of 3,540 mBlock programs, we find a total of 6,129 instances of bug patterns, 592 code smells and 14,495 code perfumes. This demonstrates the potential of our approach to provide feedback and assistance to learners and educators alike for their mBlock robot programs.

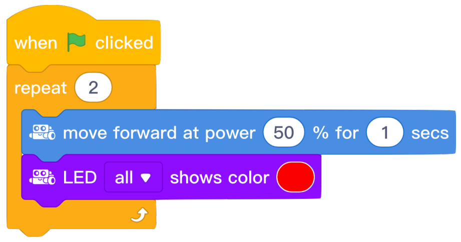
CCS CONCEPTS

• **Social and professional topics → K-12 education**; **Software engineering education**; • **Software and its engineering → Visual languages**.

KEYWORDS

mBlock, Robot, Block-based programming, Linting, Code quality

ACM Reference Format:
Florian Obermüller, Robert Pernerstorfer, Lisa Bailey, Ute Heuer, and Gordon Fraser. 2022. Common Patterns in Block-Based Robot Programs. In *Proceedings of the 17th Workshop in Primary and Secondary Computing Education (WiPSCE '22), October 31-November 2, 2022, Morschach, Switzerland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3556787.3556859



```
when 🏁 clicked
repeat (2)
    move forward at power (50) % for (1) secs
    LED [all ▾] shows color (🟥)
```

Bug Pattern LED Off Missing found: The LEDs on your robot are still turned on after the program has stopped. Add an LED Off block at the end of your program.

Figure 1: Example mBlock program for an mBot robot, containing a bug: The robot is moved forward twice and the LEDs are set to red, but they are not turned off again. This bug is identified by our LitterBox extension and provides a textual hint.

1 INTRODUCTION

A common means to introduce young learners to programming is by using robots. Programmable robots are fun to interact with and engage learners. Their interactions with the real world through sensors and actuators rather than simulated environments make them well suited for cross-curricular activities. The programming environments commonly used for controlling such robots are based on established introductory block-based programming languages. For example, the popular Scratch [13] programming environment offers 'extensions' that add dedicated blocks for controlling different types of Lego robots (e.g., Mindstorms EV3, Boost, WeDo), and the various robots produced by MakeBlock (e.g., mBot, Codey Rocky) can be programmed in mBlock, a modified version of Scratch. These modified programming environments thus offer intuitive first steps into entry level programming and make the transition to purely computer-based programming easy [1].

Figure 1 shows an example mBlock program controlling an mBot robot: When the program is started on the computer, the robot is moved forward twice for a second at half its maximum speed, and its LEDs are set to red. Except for the two robot-specific blocks, the program itself is indistinguishable from any other standard Scratch program learners might create. However, the robot-specific blocks have real-world implications that learners need to know and comprehend in order to properly control the robots. For example, while the 'move forward' block turns off the motor after one second, the 'LED' block only turns the lights on, but not off. Consequently, the program has a bug: Executing the code will have the side-effect that the lights are constantly on while the program is running, and also remain turned on even after the execution has completed. This bug is based on a common misunderstanding of learners who start to interact with robots.

To support learners in overcoming their misconceptions and building correctly working programs, they may require assistance. While this assistance by default is provided by educators, these might be overwhelmed in a classroom of students, each with individual problems in their programs. However, many challenges in programming are repetitive and can be detected automatically in learners' programs. Different types of patterns can be identified: *Bug patterns* are code constructs that provide evidence of misconceptions and bugs; *code smells* are aspects of code that may achieve the desired output, but are ineffective or confusing in some way; and *code perfumes* provide evidence of correctly applied code constructs and idioms. For block-based programming languages like Scratch, tools identifying these types of patterns are available [4, 18]. However, existing tools and patterns do not cover the specific challenges caused by robot programming.

In this paper, we therefore aim to extend the concept of these patterns to introductory robot programming. We create a dataset of 3,540 programs written in the mBlock programming language for the popular mBot and Codey Rocky programmable robots. Using this dataset, we determine and evaluate a catalogue of bug patterns, code smells, and code perfumes, and implement them as an mBlock extension of the LitterBox [4] analysis framework. When instances of the negative code patterns are found, our implementation also automatically generates hints, providing feedback on why there is a problem and what the underlying misconception may be. Furthermore our automatically generated hints also suggest how to fix the bug or to remove the code smell. For the program in Figure 1, our LitterBox extension reports an instance of the *LED Off Missing* bug pattern, stating that the LEDs are turned on after the program ended and that the user should introduce an LED Off block to fix this problem.

Our experiments suggest that Scratch projects are structurally different from mBlock programs; they are smaller and have fewer scripts, but individual scripts can be similarly complex. We found instances for most bug patterns, code smells and code perfumes in our dataset, suggesting that the patterns are highly applicable. A manual classification also demonstrates that mBlock bug patterns are a frequent cause of failures, i.e., program states where the robot observably misbehaves. The integration of these patterns into LitterBox enables educators and learners to immediately make use of this information.

2 BACKGROUND

In order to engage young learners with the concepts of programming, two important approaches are (1) to simplify the construction of programs using block-based languages and (2) to use programmable robots. Constructing such programs for robots gives rise to specific challenges. The aim of this paper is to derive and evaluate code patterns that can help to address these challenges.

2.1 Educational Robots

A popular way to introduce children to programming is using programmable robots [15]. This has multiple reasons: First, robots offer an easy starting point as they usually can be controlled without a computer, such as the Ozobot robots [11]. Second, interacting with the environment rather than programming simulated environments on the computer can lead to higher learning motivation [19]: Students have to investigate aspects of the robots' capabilities and can then tackle real world problems, like reading from a sensor and letting the robot act accordingly. Third, programming robots leads to a combination of acquiring programming skills with other abilities like spatial thinking [10], and the use of robots may also lead to further discussion about the consequences of programming and program execution, for example as motors could be overstrained or other parts of the robot could be damaged. Finally, robots are well suited for cross-curricular activities (e.g., physics, art, physical education) due to their sensors and actuators [23]. As an example, the workings of an ultra sonic sensor can be discussed, and measured data can be used in classic tasks like calculating the speed of the robot from time and travelled way.

Educational robot programming environments are usually intended to help transition to solely computer based programming. For example, the MakeBlock line of robots and their mBlock[1] programming environment achieve this by using an extended version of the popular block based programming language Scratch [13]. mBlock uses the exact same blocks and shapes as Scratch to prevent syntactical errors for making programming more accessible for novices. The Scratch programming environment is extended with new blocks for controlling the robots' actuators and reading the sensors of the robots when connected to the computer.

Two popular types of robots compatible with mBlock are the Codey Rocky and the mBot. Both robots have two motors to move each side separately. Also, with both robots one can sense the intensity of the ambient light, display information on an LED matrix, and turn LEDs off and on. Furthermore, the mBot has an ultra sonic sensor for measuring distances and a line following sensor to detect if the robot is driving over dark or bright ground. The Codey Rocky, on the other hand, has a gyroscope, additional lights, a colour sensor as well as a potentiometer. Thus, both robots have a variety of sensors and actuators that can be read and controlled with the additional blocks of mBlock.

2.2 Patterns in Block-based Programs

Even though block-based languages are designed with the aim to make programming more accessible and intuitive to novices, it can nevertheless be challenging to assemble the blocks in a correct way that implements the desired functionality. An important means to provide feedback and support to learners, and for research or educational purposes, is to identify common patterns of blocks in the learners' programs. There are three main categories of such patterns which have been explored in the context of block-based programs.

Code smells are idioms of code that decrease the understandability of the project and increase the likelihood of bugs occurring when modifying the code [3]. A range of studies have investigated the types and occurrences of code smells in block-based languages [7, 13, 16, 26]. There is evidence that the presence of code smells hampers the ability of learners to modify the code [6], and that code smells can decrease the likelihood of projects being reused [26] in

[1]http://mblock.makeblock.com/, last accessed 01.06.22

remixes [1]. Code smells can be detected automatically using tools such as Quality Hound [25], Hairball [2], or LitterBox [4].

While code smells capture attributes that are independent of whether the affected code is correct or not, *bug patterns* refer to aspects of code that are likely to lead to undesired behaviour (i.e., *bugs* or *defects* [8]). Similar to code smells, bug patterns can be detected automatically on source code [12, 17]. Bug patterns have been shown to appear frequently in Scratch projects [5], and there are automated tools that can be used to find them, such as LitterBox [4]. Note that instances of bug patterns in code are likely candidates of bugs, but are not guaranteed to be incorrect. Determining whether a program is truly broken would require running the program and testing whether it behaves as expected. While there are approaches to do this automatically also for block-based languages (e.g., Whisker [22], Itch [9], Bastet [21]), these approaches require task-specific tests or specifications that describe the expected behaviour. In contrast, patterns are task-independent, and therefore only need to be defined once in order to enable detection on any program.

Code smells and bug patterns aim to find problems, but it is also possible to identify positive aspects of code, which may for example serve as evidence of progress or as positive feedback. Code idioms indicating good programming practices or code showing understanding of certain programming concepts are known as *code perfumes* [18], and can be seen as the opposite of code smells. Technically, matching code perfumes on block-based programs is similar to matching code smells and bug patterns, which means tools like LitterBox [4] can also detect code perfumes automatically.

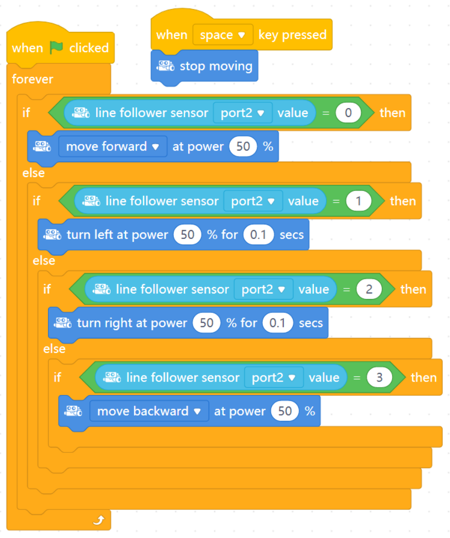
While these concepts of code patterns are well explored in the context of block-based programming, none of the existing approaches focus explicitly on the robot-specific aspects of code. The aim of this paper is to fill this gap. Since mBlock is based on the Scratch programming language and extends it by the robot functionality, we expect to find new types of all statically detectable code patterns in the programs for mBot and Codey Rocky robots.

3 PATTERNS IN MBLOCK

Along with the additional possibility of controlling robots, and thus real physical hardware, new problems can occur. These problems can arise from the physical limitations of the robots, from the way the robot software itself is implemented, or also from peculiarities in handling robots or physical hardware, e.g., reading out sensor values and reacting to them. Furthermore, there are also new situations where learners can profit from positive feedback. By reflections of students of computer science education, programming courses with children, comparing robot behaviour with known patterns in Scratch, and by our own experimentation with the robots, we discovered 26 bug patterns, three code smells, and 18 code perfumes.

3.1 Bug Patterns

A bug pattern in Scratch is a composition of blocks that are typical of buggy code, or a general buggy deviation from a correct code idiom [5]. Following this notion, robot bug patterns in mBlock are compositions of blocks that cause the robot to act incorrectly or exhibit undesirable behaviour.


```
when clicked
forever
  if < line follower sensor port2 value = 0 > then
    move forward at power 50 %
  else
    if < line follower sensor port2 value = 1 > then
      turn left at power 50 % for 0.1 secs
    else
      if < line follower sensor port2 value = 2 > then
        turn right at power 50 % for 0.1 secs
      else
        if < line follower sensor port2 value = 3 > then
          move backward at power 50 %

when space key pressed
stop moving
```

Figure 2: Example Action Not Stopped bug pattern.

Action Not Stopped: Stop commands for actuators (LED, light, matrix, motor) only switch off the corresponding actuator, but do not end the scripts in which the actuator is used. The scripts themself, possibly containing loops, continue to run. If a statement for an actuator is within a loop and this actuator is terminated by a stop command from another script, the statement will nevertheless be executed again by the loop. If all uses of an actuator are to be stopped, the scripts that use this actuator have to be stopped, too. Figure 2 shows an example of this bug pattern, where the actuators are not correctly stopped.

Actuator Deactivation Missing: In contrast to time-limited statements, some blocks only switch actuators on without switching them back off after a while. Even after all scripts have stopped, the actuators remain active. If the program code contains no scripts that turn them off permanently, the only possibility to deactivate them is by completely switching off the robot. This problem can easily be prevented with scripts that only switch on the actuators for a limited time, or by a separate stop script that deactivates them. We define several versions of this bug pattern depending on the specific actuator used:

- **LED Off Missing**
- **Light Off Missing**
- **Matrix Off Missing**
- **Motor Off Missing**

Figure 1 is an example of the LED Off Missing bug pattern.

Colour Out of Range: The defined values for colours are the integers from 0 to 255. Setting the colour to other values is possible, but values higher than 255 lead to an actual colour setting of 255, and negative values to 0. Thus, the code would not match the result, and only integers form 0 to 255 should be used.

Interrupted Loop Sensing: A typical concept of robot programming is making the robot react to specific sensor values. Using a sensor query within a loop that contains time-limited statements can cause a bug: When the sensor values are not read out frequently enough, the robot might not react to short occurrences of the relevant values. To prevent this, time-limited statements and queries concerning sensors should be in separate loops.

Low Motor Power: The electric motors of the mBot robot need a minimum amount of power to move the wheels. When the power value of a movement statement is set to less than 25%, the corresponding wheels will not turn. Programs targeting low movement speeds should be avoided or used for a Codey Rocky robot instead, which does not have a minimum threshold for the motor power.

Missing Loop Sensing: In order to make the robot react to a specific change in the values of a sensor, one must continuously read out the corresponding sensor values. When a query concerning sensors is not within a loop, it will only be executed once, which leads to the robot not reacting to later value changes. In order to read out the sensor values continuously, the query should be inside a forever loop or a loop with a stopping condition.

Motor Out Of Range: The motors can only be controlled with a maximum of 100%. It is possible to set higher power values, but in practice, the motor still does not run faster. When power values above 100% are used, the result does not match the code, since all values above 100% lead to the same speed as 100%. Analogously, this also applies to values beneath -100%. In general, only power values within a reasonable range (i.e., up to 100%) should be used.

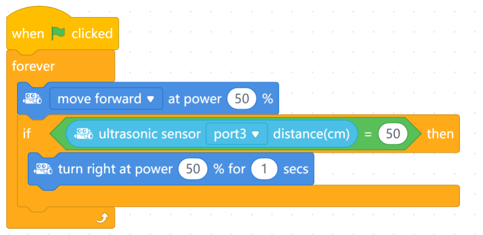
Parallel Actuator Use: When two scripts run in parallel and use the same actuators, they block each other or cancel the other use. To avoid such conflicts, different scripts should use different actuators, or one has to make sure that other scripts have finished beforehand.

Query In Loop: Physical buttons or sensors on a robot are not only activated for an instant when they are triggered. Instead, buttons remain pressed and sensors return values for a period of time. When a query is used as a condition of a fast loop (i.e., a loop without time or wait statements), the query is repeated very frequently, in some cases even several hundred times per second. Thus, the code reacting to detected values is executed uncontrollably often and can falsify other values or states that are changed according to the query result. To prevent this, sensor queries within a loop that change values should be avoided or secured by waiting statements.

Sensor Equals Check: The values of the robots' sensors are not rounded and are therefore rarely at a single exact value. If a sensor is compared to an exact value in the code, the probability of this state never occurring is very high. Therefore, when querying the sensors, one should compare to a range of values instead of exact values. An example of this bug pattern is shown in Figure 3. The line-following sensor is an exception, as the values for this sensor are just integers from 0 to 3, allowing an exact comparison.

Several Launches: When used with the 'upload mode', the mBot robot cannot execute scripts in parallel, but only sequentially. In addition, the board launch event is the only event that works in upload mode. If two board launch scripts are programmed in parallel, they are only executed one after the other, or — in the case of forever loops — sometimes not at all. For parallel programming, either the 'live mode' or a Codey Rocky robot must be used.

Stuttering Action: In live mode, the calculation of all scripts is taken over by the computer, and the corresponding commands are sent to the robot individually. This runs noticeably slower than



```
when 🏴 clicked
forever
  move forward ▾ at power (50) %
  if < ultrasonic sensor port3 ▾ distance(cm) = (50) > then
    turn right at power (50) % for (1) secs
```

Figure 3: Example Sensor Equals Check bug pattern.

when code is uploaded to the robot. If individual time-limited statements are in a loop in live mode, they are not executed smoothly one after the other. Instead, the code stops briefly between each block. To prevent this, time-limited statements in loops should generally be avoided and replaced by constant move blocks guarded by a query in order to have a stopping condition.

Useless Sensing: If the value range of a query involving a specific type of sensor is out of the allowed range, then coresponding conditions will either always be true, or always be false. We distinguish several variants of this pattern depending on the sensor involved:

- **Useless Battery Sensing:** The value range for Battery Sensing is from 0 to 100.
- **Useless Colour Sensing:** The value range for Colour Sensing is from 0 to 255.
- **Useless Distance Sensing:** The value range for Distance Sensing is from 3 to 400.
- **Useless Light Sensing:** The values for Light Sensing is from 0 to 100 for the Codey Rocky and from 0 to 1020 for the mBot.
- **Useless Line Sensing:** The values for Line Sensing are the integers from 0 to 3.
- **Useless Loudness Sensing:** The value range for Loudness Sensing is from 0 to 100.
- **Useless Pitch Angle Sensing:** The value range for Pitch Angle Sensing is from -180 to 180.
- **Useless Potentiometer Sensing:** The value range for Potentiometer Sensing is from 0 to 100.
- **Useless Roll Angle Sensing:** The value range for Roll Angle Sensing is from -90 to 90.
- **Useless Shaking Sensing:** The value range for Shaking Sensing is from 0 to 100.

Waiting Aborted: mBlock offers blocks with a time limit, which activate a specific actuator for the specified time. When a program with such blocks is uploaded to the robot, the time-limited statement is split into three separate parts: activation, waiting, and deactivation. Executing a parallel script which cancels all scripts on a Codey Rocky robot can lead to the following problem: When the scripts are cancelled during the waiting phase of the time-limited statement, the waiting itself is cancelled, but also, the deactivation will no longer be carried out. This can lead to motors running indefinitely or lights not being switched off. The bug can be prevented by using a detailed stop script that stops all affected actuators before the scripts are terminated.

3.2 Code Smells

A code smell in mBlock is a code idiom that increases the probability of errors in a program or decreases the readability of the code by using unexpected values when addressing sensors and actuators.

Negative Motor Power: Using negative motor power values is possible and leads to the robot changing its direction. I.e., when using negative values for moving forward, the robot moves backward, and vice versa. The same applies when turning left or right. Since using negative values reduces the code readability, one should rather use the corresponding blocks (e.g., move backward with positive value instead of move forward with negative value).

Non-effective Modification: Changing the settings of a robot attribute several times in a row can lead to the robot behaving differently than expected: Without waiting phases in between the modifications, the attributes will be set to the next value so fast that one cannot see any effect. Only the last setting will be visible. In order to notice every single modification, time-limited statements or waiting blocks in between the attribute settings have to be used.

Non-effective Time Limit: When the time value of a time-limited block is set to 0, the execution of the block will have no effect. Writing unnecessary code is bad for readability and should be avoided.

3.3 Code Perfumes

Code perfumes for mBlock are aspects of code that correctly apply concepts related to robot use, in particular correct use of sensory data and actuators.

Colour Usage: Appropriate values for colours are integers from 0 to 255.

Correct Sensing: Queries involving a sensor should use a valid value range. This indicates comprehension of the sensing concept, which is frequently used in robot programming. Depending on the sensor used, we distinguish several variants of this code perfume:

- **Battery Sensing:** The value range for Battery Sensing is from 0 to 100.
- **Colour Sensing:** The value range for Colour Sensing is from 0 to 255.
- **Distance Sensing:** The value range for Distance Sensing is from 3 to 400.
- **Light Sensing:** The value range for Light Sensing is from 0 to 100 on the Codey Rocky and from 0 to 1020 on the mBot.
- **Line Sensing:** The values for Line Sensing are the integers from 0 to 3.
- **Loudness Sensing:** The value range for Loudness Sensing is from 0 to 100.
- **Pitch Angle Sensing:** The value range for Pitch Angle Sensing is from -180 to 180.
- **Potentiometer Sensing:** The value range for Potentiometer Sensing is from 0 to 100.
- **Roll Angle Sensing:** The value range for Roll Angle Sensing is from -90 to 90.
- **Shaking Sensing:** The value range for Shaking Sensing is from 0 to 100.

Correct Actuator Deactivation: When using blocks that activate an actuator, one must be aware of the necessity of also deactivating them. Writing a separate script for turning the actuators off is often useful and shows that this robot specific usage of actuators has been understood. We define several versions of this code perfume depending on the specific actuator used:

- **LED Off**
- **Light Off**
- **Matrix Off**
- **Motor Off**

Loop Sensing: In order to make the robot react to a specific change of a sensor's value, one must continuously read the corresponding sensor values. Using queries concerning sensors within a loop indicates the comprehension of this robot typical concept of sensing.

Motor Usage: The motors of the robots can be controlled with a minimum of 0% and a maximum of 100%. When using the mBot robot, values beneath 25% have the same effect as 0%. Therefore, appropriate values range from 0 to 100 for the Codey Rocky robot, and from 25 to 100 for the mBot robot.

Parallelisation: Writing several scripts with the same hat block can be indicative of attempts to implement independent subtasks at a higher readability level.

4 EXPERIMENTAL SETUP

To evaluate the different types of code patterns for mBlock projects, we empirically investigated the following research questions:

- **RQ1:** How does mBlock code compare to Scratch code?
- **RQ2:** How common are robot bug patterns, code smells and code perfumes in mBlock programs?
- **RQ3:** How severe are the bug patterns found?

4.1 Analysis Tool

In order to study the occurrence of the patterns listed in Section 3 in mBlock programs, we extended the LitterBox [4] tool, which was originally intended for Scratch projects (see Section 2). LitterBox handles the analysis of Scratch programs by automatically converting a project into an abstract syntax tree (AST), and then checking for the presence of block combinations utilising a visitor pattern. Each bug pattern, code perfume and code smell has its own visitor looking for the block combination defining the code pattern. Since mBlock is a fork of Scratch adding robot functionalities, the extension of LitterBox required us to implement handling for all mBlock blocks in the parser, so that programs for Codey Rocky and mBot can be represented as ASTs (which then consist of both standard Scratch nodes as well as mBlock-specific nodes). For each of the finders described in Section 3 we then implemented an AST visitor that traverses the blocks of a program and reports all matches found. Besides this AST visitor, each pattern also consists of a textual hint defined in multiple languages, which can be shown to a user checking their program for patterns.

4.2 Dataset

4.2.1 mBlock dataset. As subjects of our study we created a data set of 28,192 mBlock programs by mining all publicly shared projects from the mBlock website[2] until the first quarter of 2021. Out of all these projects, 329 resulted in parsing exceptions when attempting to apply LitterBox. Of these, 32 were empty files without any code,

[2]https://planet.mblock.cc/, last accessed 01.06.22

while the remaining 297 projects contain robot features not supported by our extension (which focuses on Codey Rocky and mBot features). This leaves a data set of 27,863 programs for analysis.

Scratch programs generally organise code in terms of different actors (i.e., sprites, background). mBlock programs add dedicated actors for robots which we can use to identify which type of robot an mBlock program is intended for. Out of the 27,863 programs, 16,569 contain a Codey Rocky or mBot robot (and sometimes more than one). The remaining 11,294 programs either contain code for other robots, or mostly are regular Scratch programs, since mBlock can also be used as a Scratch programming environment. Since the mBlock programming environment used to add a Codey Rocky robot actor by default in the past, we further filtered the dataset by removing all programs where the mBot and Codey Rocky actors contain no code. This leaves a final dataset of 3,540 relevant projects with a total of 529 Codey Rocky robots and 3,023 mBot robots, including 27 projects utilising both robots. For the remainder of this paper, the 3,540 mBlock projects are used as the robot dataset.

4.2.2 Scratch dataset. In order to compare robot code with regular Scratch code, we created a comparable set of Scratch programs in addition to the existing robot set. We used the Scratch REST-API to mine a dataset of more than 1 million projects, of which we randomly sampled 3,540 non-empty, non-remixed programs.

4.3 Methodology

4.3.1 RQ1: mBlock vs. Scratch differences. We applied LitterBox on the Scratch and the mBlock dataset and collected all information about bug patterns, code smells, code perfumes, and code metrics available in LitterBox, including those added by our mBlock extension. To answer RQ1 we characterise the differences between regular Scratch programs and mBlock programs in terms of the code metrics, compare the programs in terms of traditional patterns found, as well as overall number of findings reported by LitterBox. For all significance tests we used a non-parametric test, the Mann-Whitney-U test [14], with $\alpha = 0.05$, as this test is designed for independent samples. The effect sizes are calculated using Vargha-Delaney's $\hat{A}_{12}$ [27]. In our context, the $\hat{A}_{12}$ is an estimation of the probability that, if we extract a metric using LitterBox on an mBlock program, we will obtain higher values than extracting the metric on a Scratch program. If mBlock and Scratch programs are equivalent with respect to a metric, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{12} = 1$ means that the metric is higher on all mBlock programs, a low value $\hat{A}_{12} = 0$ means that the metric is higher on all Scratch programs.

4.3.2 RQ2: mBlock pattern occurrences. To answer this research question, we consider the findings reported for all the mBlock-specific patterns on the mBlock dataset. For each pattern, we inspect the total number of instances found, how many programs are affected, and how the patterns relate to program complexity, as measured by different metrics such as the weighted method count (i.e., sum of cyclomatic complexities of all scripts).

4.3.3 RQ3: mBlock bug pattern severity. To answer RQ3, we specifically consider the mBlock bug patterns. For each bug pattern, we randomly selected 10 projects that contain at least one such bug, or all bugs if there are less than 10 instances overall. Then, two authors manually classified the projects into the three categories *failures* (the bug pattern causes incorrect program behaviour), *not executed* (the bug pattern may represent incorrect code, but this code cannot be executed), and *false positives* (the bug pattern does not affect the execution). The classification requires executing the programs using a robot. In case of disagreement between the two raters, the cases were discussed among the authors until a consensus was found.

Table 1: Mean values of metrics for Scratch and mBlock datasets compared.

	mBlock	Scratch	p-value	$\hat{A}_{12}$
#Blocks	51.25	127.48	<0.001	0.47
WMC	8.97	29.88	<0.001	0.38
#Scripts	3.89	13.84	<0.001	0.32
Longest Script	15.52	16.38	<0.001	0.53
Most Complex Script	3.54	4.18	0.4	0.51

4.4 Threats to Validity

External validity: Although we used a large dataset of mBlock programs, and an equally large set of Scratch programs, results may not generalise to other programs. In particular, our data mining approach can only download publicly shared projects, and incomplete programs may not be shared. *Internal validity*: We thoroughly tested our implementation to avoid bugs in the implementation. To reduce the threat of misclassification for RQ3, two authors independently classified findings. *Construct validity*: While we measured the frequency of all patterns, we only evaluated severity for bug patterns, and generally did not evaluate their effects on learners.

5 RESULTS

5.1 RQ1: How Does mBlock Code Compare to Scratch Code?

5.1.1 Size and Complexity. Table 1 compares the programs in our mBlock and Scratch datasets in terms of size and complexity metrics: On average, Scratch projects have significantly more blocks than mBlock programs. This is also confirmed by the distribution of sizes shown in Fig. 4, with Scratch programs having up to 6,790 blocks, while no mBlock program is larger than 2,923 blocks. The overall larger complexity is also confirmed by the mean weighted method count (WMC), which again is significantly higher for Scratch programs (average of 29.88) compared to mBlock (8.97 on average). This may be attributed to the limited scope of robot programs: There is only so much you can do with the same set of sensors and actuators, while the boundary of possibilities and complexities in Scratch will not be reached so soon.

One factor leading to fewer scripts in robot programs may be that mBot only allows one script when working in upload mode (cf. *Several Launches* bug pattern). This is confirmed by the statistically significant difference in the number of scripts in Table 1 and when comparing to mBot programs only. While we also observe a significant difference in the length of the longest script (Table 1), the cyclomatic complexity of the most complex script per project is *not* significantly different ($p = 0.4$). Consequently, it seems that the more complex nature of Scratch projects lies in parallelism, while individual scripts can be just as complex in mBlock.

5.1.2 Bug Patterns. An alternative way to understand the differences between the programs is by considering how many general Scratch patterns (i.e., excluding the new mBlock-patterns) are found in the different types of programs. In the total of 3,540 programs each, 4,942 bug patterns were found for the mBlock projects and 18,698 for the Scratch dataset, which is a large and statistically significant difference ($p < 0.001$, $\hat{A}_{12} = 0.33$). Since Scratch programs contain more blocks, there naturally are more possibilities for bug patterns, but this difference remains significant even when normalising by the number of blocks in a program ($p < 0.001$, $\hat{A}_{12} = 0.32$). On the one hand, this suggests that mBlock programs only use a subset of the functionalities of Scratch: For example, some Scratch bug patterns (e.g., *Message Never Received*, *Forever Inside If*) [4] occur also in mBlock programs, while aspects like cloning of sprites generally are not contained in mBlock programs. On the other hand, mBlock programs add new functionality that may not be captured by the existing bug patterns. When considering the total number of bug patterns, including the mBlock-specific ones introduced in this paper, the difference between the datasets is no longer significant (p=0.126). The total numbers are shown in Table 2. This confirms the need for robot-specific bug patterns, and demonstrates that our patterns capture the robot functionality well.

5.1.3 Code Smells. The comparison in terms of Scratch code smells paints a similar picture: There are 23,933 in the mBlock projects and 43,996 for Scratch, which is a significant difference ($p < 0.001$, $\hat{A}_{12} = 0.43$) even when normalising for block count ($p = 0.003$, $\hat{A}_{12} = 0.48$). Unlike for bug patterns, however, even when considering also the mBlock-specific code smells, the difference remains significant ($p < 0.001$, $\hat{A}_{12} = 0.43$). This is because code smells tend to be more concerned with the general structure of programs than with specific functionality. While code smells like *Code Clones*, *Duplicated Script* and *Long Script* [4] also occur in mBlock programs, the simpler structure of mBlock programs provides less opportunities for such smells. A noteworthy exception for mBlock is *Empty Sprite*, since mBlock initialises an empty graphical sprite by default, even though it is not needed for robots.

5.1.4 Code Perfumes. Code perfumes are similarly imbalanced, with a total of 30,296 Scratch perfumes for mBlock programs, and 135,494 for Scratch programs ($p < 0.001$, $\hat{A}_{12} = 0.34$, and $p < 0.001$, $\hat{A}_{12} = 0.29$ when normalising by number of blocks). Like code smells, many code perfumes are concerned with control flow aspects that seem to occur less frequently in mBlock programs, as also suggested by the higher complexity of Scratch programs (Table 1). The imbalance remains even when including all new code perfumes ($p < 0.001$, $\hat{A}_{12} = 0.45$). This is because the mBlock specific perfumes like the four *Off* variants are only needed once per project and thus do not increase the count of perfumes by much. Furthermore, a Scratch project with multiple sprites may need multiple collision checks, whereas Codey Rocky and mBot programs usually just need a single check for obstacles ahead.

Summary (*RQ1*) Scratch and mBlock programs are significantly different regarding their size and complexity. Robot-specific patterns are needed to analyse mBlock programs as thoroughly as Scratch projects.

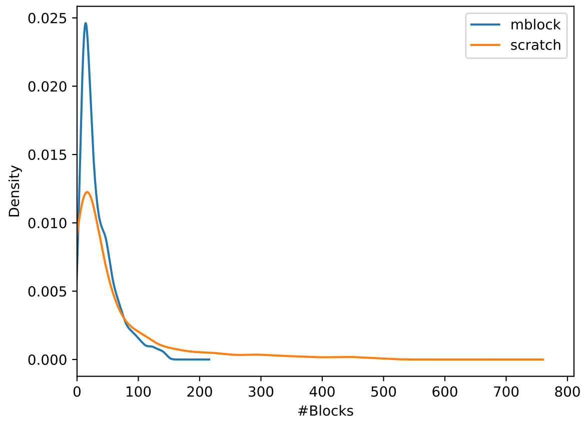


Figure 4: Density of the block count compared in Scratch and mBlock programs.

Table 2: Number of pattern instances found using only the Scratch finders or all incl. mBlock on both the Scratch and mBlock datasets.

	Scratch		All	
Patterns	mBlock	Scratch	mBlock	Scratch
Bug patterns	4,942	18,698	11,071	18,698
Code smells	23,933	43,996	24,525	43,996
Code perfumes	30,296	135,494	44,791	135,494

5.2 RQ2: How Common are Robot Bug Patterns, Code Smells and Code Perfumes in mBlock Programs?

5.2.1 Bug Patterns. We found instances for 19 of the 26 robot bug patterns in the dataset of random mBlock projects. In total, there are 6,129 bug pattern instances, and 1,903 projects contain at least one bug pattern. Table 3 summarises the number of bug pattern instances found for each type, the number of projects containing at least one instance of the respective pattern, and the average weighted method count of these projects. Note that one project may contain more than one type of bug pattern.

The bug patterns that were not found in the dataset are all related to sensors that are only found on the Codey Rocky, for which the dataset only contains 529 projects. Even within these projects, these sensors are only rarely used, as shown in Table 4: These sensors were mostly used for calculations or to show the value on the LED Matrix, but not in a way that the finders would check. However, since we found similar bug patterns for other sensors like *Useless Distance Sensing*, we expect that a growing community of Codey Rocky users will lead to all sensors being used in the future.

The most frequent bug pattern is *LED Off Missing* with 1,516 instances. Combined with the frequent other *Off Missing* bug patterns, it appears that beginners do not pay attention to returning the robot to a neutral state when the program has finished execution. Note that *Light Off Missing* is an exception here with only 12

Table 3: Number of bug pattern instances found in total and number of projects containing the bug pattern.

Bug Pattern	# Patterns	# Projects	Avg. WMC
Action Not Stopped	117	77	34.34
Colour Out Of Range	9	7	4.86
Interrupted Loop Sensing	1,287	319	14.34
LED Off Missing	1,516	675	11.37
Light Off Missing	12	9	5.56
Low Motor Power	348	90	11.38
Matrix Off Missing	694	478	13.34
Missing Loop Sensing	278	126	12.20
Motor Off Missing	491	364	11.55
Motor Out Of Range	230	68	9.53
Parallel Actuator Use	887	473	16.21
Query In Loop	26	8	49.88
Sensor Equals Check	70	34	17.71
Several Launches	52	52	18.85
Stuttering Action	89	52	13.02
Useless Battery Sensing	0	0	
Useless Colour Sensing	0	0	
Useless Distance Sensing	10	7	8.29
Useless Light Sensing	1	1	12.00
Useless Line Sensing	3	3	12.00
Useless Loudness Sensing	0	0	
Useless Pitch Angle Sensing	0	0	
Useless Potentiometer Sensing	0	0	
Useless Roll Angle Sensing	0	0	
Useless Shaking Sensing	0	0	
Waiting Aborted	9	9	10.11
Total	6,129	1,903	11.39

Table 4: Number of projects using a specific sensor.

Sensor	# Projects
Battery Level	11
Colour Detection	3
Gyro	4
Light Intensity	292
Line Following	225
Loudness	11
Potentiometer	20
Shaking	5
Ultra Sonic	736

instances found, but again this may be caused by the low number of Codey Rocky projects. This conjecture is further supported when considering the number of projects containing these bug patterns (Column 3 in Table 3): Here *LED Off Missing* is also ranked first (675) followed by *Matrix Off Missing* (478) and *Parallel Actuator Use* (473). Furthermore, the remaining non Codey Rocky exclusive bug pattern concerning the switching off of actuators (i.e., *Motor Off Missing*) come in at fourth place with 364 projects exhibiting it.

The second and third most frequent bug patterns are *Interrupted Loop Sensing* (1,287) and *Parallel Actuator Use* (887). Both of these can come from a wide range of blocks, and the causes — either using a timed block that interrupts a sensing process or an accidental use of the same actuator at the same time — are easy to create.

Table 5: Number of code smell instances found in total and number of projects containing the code smell.

Code Smell	# Patterns	# Projects	Avg. WMC
Negative Motor Power	257	71	11.42
Non-effective Modification	323	86	14.22
Non-effective Time Limit	12	8	5.38
Total	592	157	11.34

The least common bug patterns (except for those not occurring at all) are *Useless Light Sensing* (1) and *Useless Line Sensing* (3). The Light Intensity and Line Following sensors are rather easy to use, as Light Intensity has the biggest range of all sensors and Line Following uses just four integers instead of a range. Consequently, it is more difficult to make mistake here.

The average complexity of the projects containing bug patterns (column WMC in Table 3) reveals that the least complex projects are found for *Colour Out of Range* (4.86), *Light Off Missing* (5.56), *Useless Distance Sensing* (8.29) and *Motor Out of Range* (9.53). Out of these, *Colour Out of Range*, *Light Off Missing* and *Motor Out of Range* relate to very basic behaviour that can be implemented even in very small projects at early stages of programming.

The bug patterns appearing in the most complex projects are *Query in Loop* (49.88) and *Action Not Stopped* (34.34). *Query in Loop* requires variables, which are a rather advanced concept and are not as frequently used in mBlock as they are in text-based programming languages. The same holds true for *Action Not Stopped*, as this pattern needs multiple scripts and control structures, which themselves increase the WMC.

5.2.2 Code Smells. We found instances for all code smells in the dataset. In total, there are 592 code smell instances, and 157 programs contained at least one code smell. Table 5 shows the numbers of code smell instances found for each type, the number of projects containing at least one instance of the respective code smell, and the average weighted method count of these projects. Again one project may contain more than one type of code smell.

For all three aspects (i.e., number of pattern instances, number of projects showing the pattern and average WMC) the ranking is the same, with *Non-effective Modification* followed by *Negative Motor Power* and *Non-effective Time Limit. Non-effective Modification* has a wide range of blocks that can cause the smelly situation. The large number of *Negative Motor Power* code smells may be caused by copy&paste, as it may be quicker to copy a forward block and enter a negative number, rather than looking for the backward block.

The small number of *Non-effective Time Limit* code smells is likely explained by useless blocks quickly cluttering the program, so that they are frequently removed. The low average WMC for this code smell also suggests that the projects using blocks without effect are rather simple and may be from beginners in their first projects trying different values and experimenting with blocks.

5.2.3 Code Perfumes. We found instances for 16 of the 18 code perfumes defined in Section 3. In total there are 14,495 code perfume instances, and 2,284 projects containing at least one code perfume.

Table 6: Number of code perfume instances found in total and number of projects containing the code perfume.

Code Perfume	# Patterns	# Projects	Avg. WMC
Battery Sensing	1	1	9.00
Colour Sensing	0	0	
Colour Usage	829	291	12.63
Distance Sensing	1,204	548	11.90
LED Off	9	9	7.56
Light Off	0	0	
Light Sensing	259	178	12.89
Line Sensing	584	138	25.72
Loop Sensing	573	272	9.36
Loudness Sensing	12	8	13.75
Matrix Off	2	2	9.50
Motor Off	43	43	9.19
Motor Usage	10,896	1,815	9.64
Parallelisation	72	40	14.65
Pitch Angle Sensing	1	1	40.00
Potentiometer Sensing	5	3	11.67
Roll Angle Sensing	2	1	102.00
Shaking Sensing	3	3	22.00
Total	14,495	2,284	9.44

Analogous to the other code patterns, Table 6 shows the number of code perfume instances found for each type, the number of projects containing at least one instance of the respective code perfume, and the average weighted method count of these programs. Once again, a project may contain more than one type of code perfume.

Two code perfumes were not found in the dataset: The *Colour Sensing* code perfume depends on the Colour Detection sensor, which is rarely used (see Table 4). For *Light Off* the corresponding bug pattern was also rare, and the likely reason is that there are only 529 Codey Rocky projects.

The most frequent code perfume is *Motor Usage* with 10,896 instances. It is followed by *Distance Sensing* (1,204) and *Colour Usage* (829). Both perfumes concerning the correct usage of actuators in the top three represent the easiest and most basic way of working with robots. *Distance Sensing* is also to be expected as it relates to the most used sensor. The ranking is the same when considering the number of projects in Table 4.

Battery Sensing and *Pitch Angle Sensing* occurred only once each. These two perfumes, as well as other related code perfumes such as *Roll Angle Sensing* (2), *Shaking Sensing* (3) and *Potentiometer Sensing* (5), all are based on sensors which are not used frequently in our dataset (see Table 4). *Matrix Off* (2) and and *LED Off* (9) are also relatively rare; the bug patterns which are the counter pieces to these two code perfumes are among the most frequent bug patterns, demonstrating that returning the robot to a neutral state after program execution is not frequently done.

Considering project complexity (Table 6), *Roll Angle Sensing* (102.00) and *Pitch Angle Sensing* (40.00) are contained in only one project each, which happen to be more complex projects. The fact that *Line Sensing* is used in more complex projects (25.72) seems surprising at first, as it is a common and easy task. However, line following tasks tend to require several control structures, i.e., at least one loop and then one if block for each of the four states of the sensor; this explains the higher complexity.

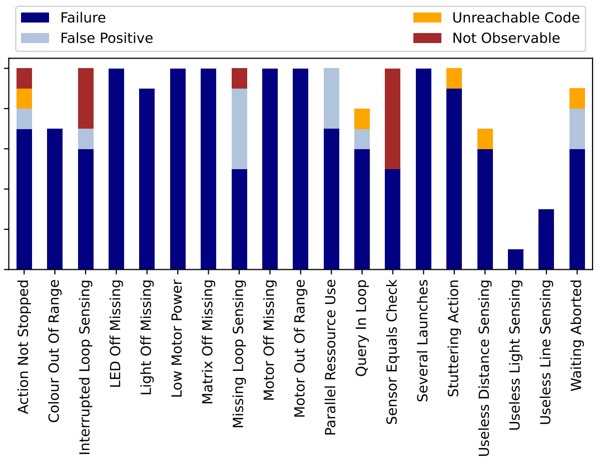


Figure 5: Results of the manual classification.

The average complexity of projects containing *Loop Sensing* (9.36) is low because most robot programs require some sensing loop in order to react to influences from the real world. The low complexity of projects containing *LED Off*, *Motor Off*, and *Matrix Off* shows that it is not difficult to correctly turn off actuators. We conjecture that actuators are rarely turned off not because it is difficult, but because users mostly are not aware it should be done.

Summary (*RQ2*) Bug patterns, code smells and code perfumes for robot projects appear frequently in mBlock programs regardless their complexity.

5.3 RQ3: How Severe are the Bug Patterns Found?

Figure 5 shows the results of the manual classification for each of the 19 bug patterns with at least one occurrence in the dataset. Out of the 164 inspected projects, 137 instances of bug patterns manifested into failures, 5 did not result in failures because the defective code was never executed, 10 were defects without visible impact, and 12 bugs were classified as false positives.

Noticeably, *LED Off Missing*, *Light Off Missing*, *Matrix Off Missing*, an *Motor Off Missing*, which are amongst the most frequent bug patterns (cf. RQ2), always lead to failures. The only fix to stop a robot in upload mode is to forcefully turn off the robot completely.

Several other bug patterns also always lead to failures (*Colour Out of Range*, *Low Motor Power*, *Motor Out of Range*, *Several Launches*, *Useless Light Sensing* and *Useless Line Sensing*). The *Stuttering Action* and *Useless Distance Sensing* bug patterns each have one case where the relevant code cannot be reached. More generally, these bug patterns demonstrate that not knowing the valid value ranges of sensors and actuators will very likely result in a defect.

The *Sensor Equals Check* bug pattern has the most cases of bugs that had no observable effects. As the language does not provide a <= operator, users try to work around this using a disjunction of two comparisons < and =. While the equality is unlikely to be true, this will usually show no effects thanks to the less-than comparison. *Interrupted Loop Sensing* produced defects where the interruptions are so short that they are difficult to observe.

As anticipated, we also found some cases of false positives, where the programmer used the mechanism we consider as a bug on purpose. *Missing Loop Sensing* has the highest false positive count (4), here the sensor value at program start is intentionally used only once to branch off into different behaviour. In principle, false positives could be avoided by refining the implementations of the bug patterns to accommodate for these exceptions.

Summary (*RQ3*) When bug patterns are executed, they frequently result in failures, but dead code and safety measures may prevent observable failures.

6 CONCLUSIONS

Patterns provide a common vocabulary for communicating about code. In this paper we demonstrated that mBlock projects are structurally different from Scratch programs, and need their own patterns that can deal with the challenges and possibilities sensors and actuators of robots provide. To this end we introduced and empirically evaluated a new catalogue of 26 bug patterns, three code smells and 18 code perfumes in mBlock. Our evaluation found occurrences for all code smells and almost all bug patterns and perfumes, which shows that the concept of bug patterns can be successfully transferred to mBlock. Furthermore, while some patterns like *Several Launches* are bound to the mBlock environment, patterns based on sensor values and motor power may also be relevant for other block based robot programming languages.

An important next step will be to study the effects of these positive and negative code patterns and the corresponding hints on the learning success of novice programmers, as well as guidelines for instructors on how to teach students about these patterns. In the future it would be interesting to see if the bug patterns result from misconceptions in programming as some seem to be a symptom of those [20, 24]. Our extended version of LitterBox is available at:

https://github.com/se2p/LitterBox

ACKNOWLEDGMENTS

This work is supported by the Federal Ministry of Education and Research through the projects 01JA2021 (primary::programming) and 01JA1924 (SKILL.de) as part of the "Qualitätsoffensive Lehrerbildung", a joint initiative of the Federal Government and the Länder. The authors are responsible for the content of this publication.

REFERENCES

[1] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (May 2017), 72–80. https://doi.org/10.1145/3015455

[2] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. 2013. Hairball: Lint-inspired static analysis of scratch projects. *SIGCSE 2013 - Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 215–220. https://doi.org/10.1145/2445196.2445265

[3] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.

[4] Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. 2021. LitterBox: A Linter for Scratch Programs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 183–188. https://doi.org/10.1109/ICSE-SEET52601.2021.00028

[5] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. 2020. Common Bugs in Scratch Programs. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. 89–95. https://doi.org/10.1145/3341525.3387389

[6] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *IEEE International Conference on Program Comprehension (ICPC)*. 1–10. https://doi.org/10.1109/ICPC.2016.7503706

[7] Felienne Hermans, Kathryn T. Stolee, and David Hoepelman. 2016. Smells in Block-Based Programming Languages. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (Cambridge, United Kingdom, 2016-09). IEEE, 68–72. https://doi.org/10.1109/VLHCC.2016.7739666

[8] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (Dec. 2004), 92–106. https://doi.org/10.1145/1052883.1052895

[9] David E Johnson. 2016. ITCH: Individual Testing of Computer Homework for Scratch Assignments. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 223–227.

[10] Sung Eun Jung and Eun-sok Won. 2018. Systematic Review of Research Trends in Robotics Education for Young Children. *Sustainability* 10, 4 (2018).

[11] Nina Körber, Lisa Bailey, Luisa Greifenstein, Gordon Fraser, Barbara Sabitzer, and Marina Rottenhofer. 2021. *An Experience of Introducing Primary School Children to Programming Using Ozobots (Practical Report)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3481312.3481347

[12] P. Louridas. 2006. Static code analysis. *IEEE Software* 23, 4 (2006), 58–61. https://doi.org/10.1109/MS.2006.114

[13] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)* 10 (11 2010), 16.

[14] Henry B. Mann and Donald R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

[15] Monica M. McGill and Adrienne Decker. 2020. Tools, Languages, and Environments Used in Primary and Secondary Computing Education. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 103–109. https://doi.org/10.1145/3341525.3387365

[16] Jesús Moreno-León and Gregorio Robles. 2014. Automatic detection of bad programming habits in scratch: A preliminary study. In *IEEE Frontiers in Education Conference (FIE) Proceedings*. 1–4. https://doi.org/10.1109/FIE.2014.7044055

[17] J. Novak, A. Krajnc, and R. Žontar. 2010. Taxonomy of static code analysis tools. In *The 33rd International Convention MIPRO*. 418–422.

[18] Florian Obermüller, Lena Bloch, Luisa Greifenstein, Ute Heuer, and Gordon Fraser. 2021. *Code Perfumes: Reporting Good Code to Encourage Learners*. Association for Computing Machinery. https://doi.org/10.1145/3481312.3481346

[19] LH Peng, MH Bai, and I Siswanto. 2020. A study of learning motivation of senior high schools by applying unity and mblock on programming languages courses. In *Journal of Physics: Conference Series*, Vol. 1456. IOP Publishing, 012037.

[20] Juha Sorva. 2018. *Misconceptions and the Beginner Programmer.*

[21] Andreas Stahlbauer, Christoph Frädrich, and Gordon Fraser. 2020. Verified from Scratch: Program Analysis for Learners' Programs. In *In Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE.

[22] Andreas Stahlbauer, Marvin Kreis, and Gordon Fraser. 2019. Testing scratch programs automatically. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 165–175.

[23] Amanda Sullivan and Marina Umaschi Bers. 2016. Robotics in the early childhood classroom: learning outcomes from an 8-week robotics curriculum in prekindergarten through second grade. *International Journal of Technology and Design Education* 26, 1 (2016), 3–20.

[24] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming Misconceptions for School Students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 151–159. https://doi.org/10.1145/3230977.3230995

[25] Peeratham Techapalokul and Eli Tilevich. 2017. Quality Hound — An online code smell analyzer for scratch programs. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 337–338.

[26] Peeratham Techapalokul and Eli Tilevich. 2017. Understanding Recurring Quality Problems and Their Impact on Code Sharing in Block-Based Software. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 43–51.

[27] Andras Vargha and Harold Delaney. 2000. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics - J EDUC BEHAV STAT* 25 (06 2000). https://doi.org/10.2307/1165329