A Block-Based Testing Framework for Scratch

PATRIC FELDMEIER*, University of Passau, Germany

GORDON FRASER*, University of Passau, Germany

UTE HEUER*, University of Passau, Germany

FLORIAN OBERMÜLLER*, University of Passau, Germany

SIEGFRIED STECKENBILLER*, University of Passau, Germany

Block-based programming environments like SCRATCH are widely used in introductory programming courses. They facilitate learning pivotal programming concepts by eliminating syntactical errors, but logical errors that break the desired program behaviour are nevertheless possible. Finding such errors requires *testing*, i.e., running the program and checking its behaviour. In many programming environments, this step can be automated by providing executable tests as code; in SCRATCH, testing can only be done manually by invoking events through user input and observing the rendered stage. While this is arguably sufficient for learners, the lack of automated testing may be inhibitive for teachers wishing to provide feedback on their students' solutions. In order to address this issue, we introduce a new category of blocks in SCRATCH that enables the creation of automated tests. With these blocks, students and teachers alike can create tests and receive feedback directly within the SCRATCH environment using familiar block-based programming logic. To facilitate the creation and to enable batch processing sets of student solutions, we extend the SCRATCH user interface with an accompanying test interface. We evaluated this testing framework with 28 teachers who created tests for a popular SCRATCH game and subsequently used these tests to assess and provide feedback on student implementations. An overall accuracy of 0.93 of teachers' tests compared to manually evaluating the functionality of 21 student solutions demonstrates that teachers are able to create and effectively use tests. A subsequent survey confirms that teachers consider the block-based test approach useful.

CCS Concepts: • **Applied computing** → **Interactive learning environments**; • **Software and its engineering** → **Visual languages**; • **Human-centered computing** → *Human computer interaction (HCI)*.

Additional Key Words and Phrases: Scratch, Block-based Programming, Automated Testing, Feedback

1 INTRODUCTION

SCRATCH makes writing code easy: There is neither a need to memorise programming commands nor programming syntax, and programs are created by simply dragging and dropping the available programming blocks to assemble

---

*Authors are listed in alphabetical order.

Authors' Contact Information: Patric Feldmeier, patric.feldmeier@uni-passau.de, University of Passau, Germany; Gordon Fraser, gordon.fraser@uni-passau.de, University of Passau, Germany; Ute Heuer, ute.heuer@uni-passau.de, University of Passau, Germany; Florian Obermüller, obermuel@fim.uni-passau.de, University of Passau, Germany; Siegfried Steckenbiller, steckenbiller@fim.uni-passau.de, University of Passau, Germany.

(a) The correct initialisation script makes the test pass.   (b) The faulty initialisation script makes the test fail.
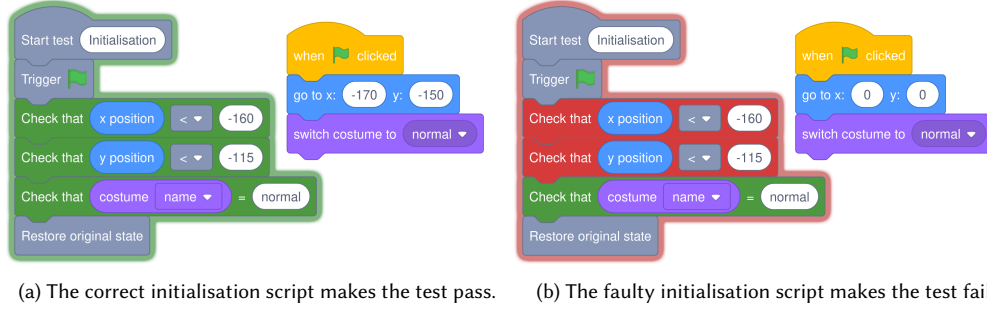
Fig. 1. A correct and a faulty variant of an initialisation script are tested.

scripts, which are syntactically correct if the blocks' shapes interlock. However, even a syntactically correct program can be functionally broken. Determining whether or not a program implements the desired functionality requires testing, i.e., running the program and validating the resulting behaviour. In other programming environments, this process can be automated by writing code that interacts with the program and checks the resulting behaviour. Running such test code provides feedback on functionality automatically, quickly, and repeatedly. In Scratch, however, testing has to be done entirely manually by interacting with the program. While this may be acceptable for learners whose focus is on being creative, it inhibits automated analyses or teachers who need to inspect programs to provide feedback.

In recognition of this gap, research prototypes for testing Scratch programs have previously been proposed [12, 20], demonstrating just how useful tests can be for Scratch, for example by automating tedious tasks such as assessing student programs [20] or providing feedback [16]. However, these testing frameworks usually require tests to be written in other languages, such as JavaScript or Python, and require additional tooling beyond the Scratch interface for execution. This, unfortunately, represents a fundamental inhibitor hampering adoption by teachers.

In this paper we therefore introduce a Scratch extension for creating and executing tests directly within Scratch. Figure 1a shows a program that sets the position of a sprite to (-170, -150) and changes the costume to *normal*, together with a corresponding block-based test. When executed, the test triggers program execution through a green flag event, subsequently checking the sprite's position and costume. The green highlighting of the test in Fig. 1a indicates that the program exhibited the expected behaviour. On the other hand, the script in Fig. 1b incorrectly sets the position to (0, 0), causing the test to fail as the position does not reach the expected state, visualised by red highlighting. Creating and running such tests is easy and provides immediate feedback within the Scratch editor, allowing students to self-evaluate their implementation with teacher-provided tests automatically. As tests are just further executable blocks, both teachers and learners are able to use these tests to receive automated feedback on a Scratch project through the colour-coded highlighting and error messages in the control interface. In detail, the contributions of this paper are as follows:

- Block-based testing: We extend Scratch with test-related blocks for creating tests, and modify Scratch to support their execution and evaluation.
- Test control interface: We extend the Scratch user interface to control test execution and help interpret test results. Tests can be run individually or collectively, on individual programs or on batches of student solutions.
- Empirical evaluation: We evaluate the block-based testing framework with 28 teachers tasked to create tests and to use these tests when assessing student code.

Our study demonstrates that teachers can write effective tests suitable for assessing student solutions, and their feedback is encouraging. Our open source implementation is available for future research and education on testing in Scratch.

2 THE SCRATCH PROGRAMMING ENVIRONMENT

SCRATCH [13] is a block-based programming environment that aims to make programming more accessible for novices, and it is one of the most popular [14] programming environments for learning. SCRATCH favours exploration over recall by displaying all blocks in a toolbox, with programs being built by visually arranging these blocks (which represent statements and expressions) into scripts. Different block shapes demonstrate which combinations are valid. For example, boolean expressions are represented as diamond blocks ◄bool►, while other expressions are rounded blocks Ⓧ, and wherever these can be inserted, there are gaps of the same shape. Statements can be stacked, such that it is only possible to create syntactically valid programs. Many statements represent high-level actions of *sprites* interacting on a *stage*, which, together with simple media integration [1], makes it easy to combine blocks to result in game-like programs.

As the underlying engine, the SCRATCH virtual machine is responsible for executing block-based programs, maintaining a representation of a program's current state and handling the event-driven nature of SCRATCH by responding to user inputs. The execution of programs is managed by a sequencing function that mimics the parallel execution of scripts through a threading model, where each script is associated with a separate thread. Whenever a script is activated, e.g., due to user input, the corresponding thread is added to the set of currently active threads. During program execution, each active thread is given permission by the sequencing function to execute the corresponding script until either its end is reached or execution halting blocks that cause the thread to yield are encountered, such as the `Wait for ❶ seconds` or `wait until ◆` blocks [4]. If the execution of a script was paused due to an execution halting block, its execution is resumed after the sequencer has processed all other currently active threads.

The different block shapes in SCRATCH prevent syntactical errors so learners can focus on what the program should do, but it can nevertheless be challenging to achieve the intended behaviour, resulting in code containing bugs. While the process of finding and fixing bugs is a crucial part of learning to code, tools can offer support to learners as well as teachers. Misconceptions of learners often manifest in bugs that follow recurring bug patterns (code idioms that are likely to be defects [11]). Bug patterns for SCRATCH have been demonstrated to occur frequently [9] and can be detected automatically by linters [8]. Such linters perform static analyses of programs, thus applying general criteria that are suitable for any SCRATCH project. However, checking specific functionality, such as whether a sprite is placed at a given location, is task-specific and also requires running the program to test its behaviour.

Existing approaches for checking the functionality of SCRATCH programs include automated testing tools like ITCH [12] and WHISKER [20], or model checking tools like BASTET [19]. WHISKER has been shown to be useful for automated grading [20], to support next-step hint generation [7, 17], and for interactive SCRATCH tutorial systems [16]. To utilise such approaches, educators need to manually create test cases in the syntax required by these tools (e.g., JavaScript), for example by describing the order in which inputs should be sent to the program under test and what the expected program response should be, or they have to formally specify expected behaviour. However, this poses a challenge for educators not well versed in the respective syntax demanded by these tools.

In this paper we therefore investigate the idea of integrating testing directly into the SCRATCH user interface. Concurrently to our work, there have been independent proposals of similar ideas: In a recent master thesis [22], the POKE extension of SCRATCH proposed a new category of blocks that enables the creation of test cases similar to the blocks we propose in this paper. An alternative proposal [15] consists of an individual new test block that demonstrates the concept of creating automated tests through custom SCRATCH blocks. Our approach is the first to offer a fully integrated, block-based test framework that makes automated and systematic testing of SCRATCH projects available without external tools to anyone being familiar with the SCRATCH programming language.
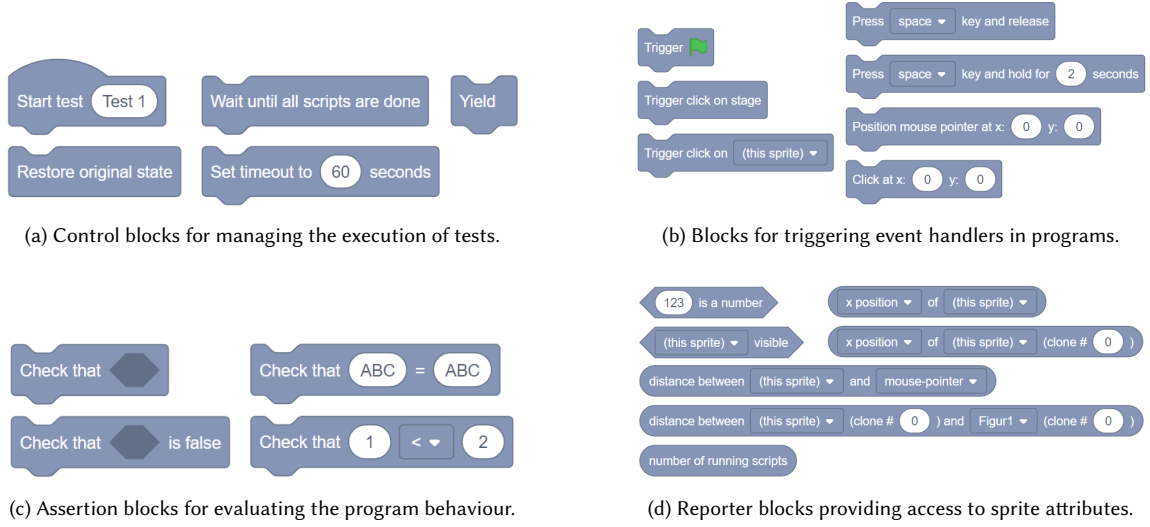
(a) Control blocks for managing the execution of tests.

(b) Blocks for triggering event handlers in programs.

(c) Assertion blocks for evaluating the program behaviour.

(d) Reporter blocks providing access to sprite attributes.

Fig. 2. New blocks introduced for testing.

3 TEST-RELATED BLOCKS

As one of the core contributions of this work, we introduce a new category of blocks that provide test-related functionality. Testing blocks encompass a diverse range of capabilities, organised into four categories: *control blocks*, which manage the execution of tests (Section 3.1); *event and input blocks*, which simulate events and user inputs to trigger corresponding event handlers (Section 3.2); *assertion blocks*, which evaluate conditions and determine the test result (Section 3.3); and *reporter blocks*, which facilitate writing tests by providing access to frequently required sprite attributes (Section 3.4).

3.1 Control Blocks

Figure 2a showcases the set of blocks added to manage the control flow of test scripts. The `Start test (Test 1)` block incorporates a wide range of functionalities: First, it enables the user to assign a specific name to the test, personalising the testing process and improving usability in the presence of multiple test scripts. Additionally, it safeguards against accidentally starting multiple tests simultaneously by preventing the start of a new test while another is currently active. The block also implicitly initiates a test timeout of five seconds to ensure that tests always end within a reasonable time and do not get stuck in infinite loops. Furthermore, the start block temporarily disables real mouse and keyboard input to prevent intentional or accidental user interference during test execution. Finally, the block is responsible for creating a snapshot of the project's current state, which can be restored after the test execution. This snapshot contains both the program state (e.g., sprite attributes and variable values) and execution state (running scripts and their execution progression).

At the end of each test script, the `Restore original state` block may be inserted to reset the project to its initial state, leveraging the snapshot taken at the start of the test execution. The restore block ensures that subsequent tests are not affected by state changes caused by previously executed tests, which may otherwise cause flaky testing behaviour [10]. Placing this block at the end of a test script is not mandatory; if omitted, the initially created snapshot is discarded and any modifications to the project state are kept, mirroring the behaviour of regular scripts that change the program state.

The `Set timeout to 60 seconds` block allows users to set or refresh the timeout that is started at the beginning of each test script. By default, the timeout is set to five seconds, which is both sufficiently long for most test scenarios and short
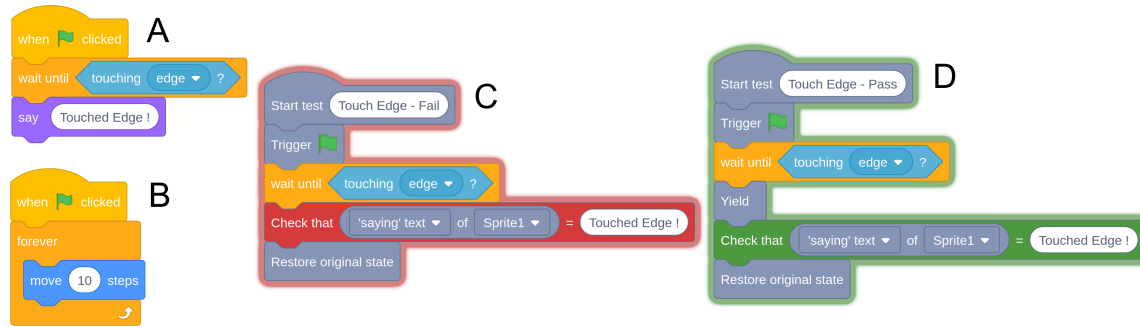
Fig. 3. The passing test D uses a *Yield* block to give execution precedence to script A.

enough for a swift execution of entire test suites. Explicitly setting the timeout is particularly valuable for tests that are expected to exceed that default timeout duration. Users can place this block at any point within the test script, enabling them to refresh the timeout during the execution of the test and assign timeouts to specific parts of their testing scripts.

The `Yield` block addresses the challenges of parallel script execution in the SCRATCH virtual machine. As explained in Section 2, the virtual machine imitates parallel processing by executing all active script threads T in sequence, with each active script t_a being executed until a yielding block is encountered or the end of the script is reached. This approach to parallelisation may however require test scripts to stop their execution temporarily in order to give precedence over the execution to other active scripts. Figure 3 illustrates this challenge with an example: consider the two tests that are supposed to assess whether a sprite displays the text *Touched Edge!* after touching the edge of the SCRATCH stage. The test script C triggers the green flag event and defers further execution of the test until `touching edge ?` evaluates to true. Since the green flag event is triggered, program script B starts to continuously move the sprite towards its heading direction, eventually causing the sprite to touch the edge of the stage after a few loop iterations. This causes a race condition: If test script C continues *before* the program script A, it immediately evaluates the assertion to check whether the sprite is displaying text, which would fail because A has not yet executed the block to display the text. However, since the execution order of scripts is unpredictable, the race condition could also have resulted in program script A taking precedence over C, resulting in a passed test. To avoid such ambiguous testing behaviour, the `Yield` block can be used to explicitly pause the execution of the test script, yielding to all other active scripts before resuming the test. Thus, inserting this deferring block after the `wait until touching edge ?` in test script D ensures that the block responsible for displaying the text is guaranteed to be executed before the test script checks whether the sprite displays the text.

In some scenarios, yielding the execution only once may not be enough. Consider the two tests in Fig. 4 that are responsible for checking whether the corresponding sprite moves to the right when the user presses the right arrow key. The left test fails because, even though triggering the script containing the movement instruction (and implicitly yielding to it), the sprite has only moved five steps to the right when the (failing) assertion block is evaluated. The sprite has not yet moved 50 steps, as the repeat block also entails an implicit yield at the end of its body, which transfers the execution priority back to the test script after its enclosed move block has been executed once. Avoiding the need to insert nine `Yield` blocks, we added the `Wait until all scripts are done` block, which defers the execution of a script until all other program scripts have stopped. Hence, we can fix the test by inserting this control block before the assertion to ensure the loop in the program script has been repeated to completion, causing the sprite to move 50 steps.
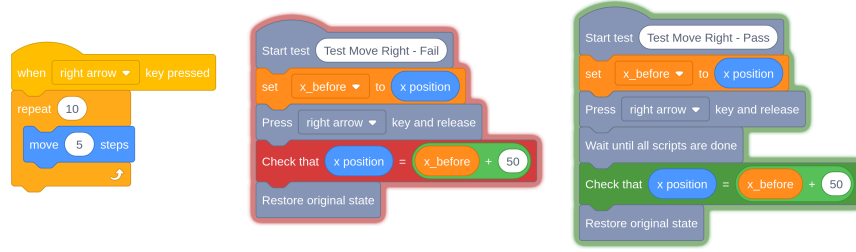
Fig. 4. The passing test waits until the script containing the movement logic has fully completed.

3.2 Trigger Blocks

As Scratch is highly event-driven, testing Scratch programs requires blocks dedicated to triggering the respective event handlers (shown in Fig. 2b) by sending corresponding events to the virtual machine. Program scripts in Scratch usually start with a hat block that listens to specific events (such as `when this sprite clicked`) and invokes successive blocks in that script upon receiving the respective event. To initiate these scripts programmatically, the trigger blocks of our extension facilitate sending user input and events to the Scratch virtual machine. Intuitively, users will expect the execution priority to switch to newly invoked scripts after a trigger block is executed. Due to the scheduling logic of the virtual machine (cf. Section 2) however, the current script would continue running. To ensure that the activated script is executed *before* the test script proceeds with evaluating the program state, every trigger block would have to be succeeded by a `Yield` block (Section 3.1). To avoid bloat in test scripts, each trigger block implicitly includes a yielding instruction, which guarantees that activated scripts are executed before the test execution is resumed and users do not have to clutter test scripts with yield blocks.

3.3 Assertion Blocks

The assertion blocks showcased in Fig. 2c provide the major purpose of every test, checking whether the program behaves as expected. Whenever those assertion blocks evaluate a condition, their colour changes to reflect the outcome: green signals that the condition has been fulfilled, while red implies that the program has failed to meet the specified condition, indicating a failure. This colour-coding system makes it straightforward for users to ascertain the reason for the success or failure of their test scripts at a glance. We consider a test passed if all its assertion blocks are satisfied and deem a test failed if any of them have a negative outcome.

In principle, all assertions could be implemented using the `Check that` block in combination with native Scratch blocks of the *operator* category. However, the additional blocks improve the ease of use and readability of tests since they embody some frequently used assertions, including comparisons of equality and numerical values. Furthermore, specialised assertions allow us to provide a more detailed explanation of why a test case failed.

3.4 Reporter Blocks

By default, the Scratch programming environment provides access to most sprite attributes, with different properties being spread across multiple blocks. For instance, the native `x position` block enables users to access the horizontal position of the current sprite, but there are no default blocks that allow users to check if a sprite is visible or is displaying some text. However, in a testing scenario, users require fast and easy access to a multitude of relevant sprite attributes, not only with regards to the currently active, but *all* sprites. In order to improve the testing experience, we added the
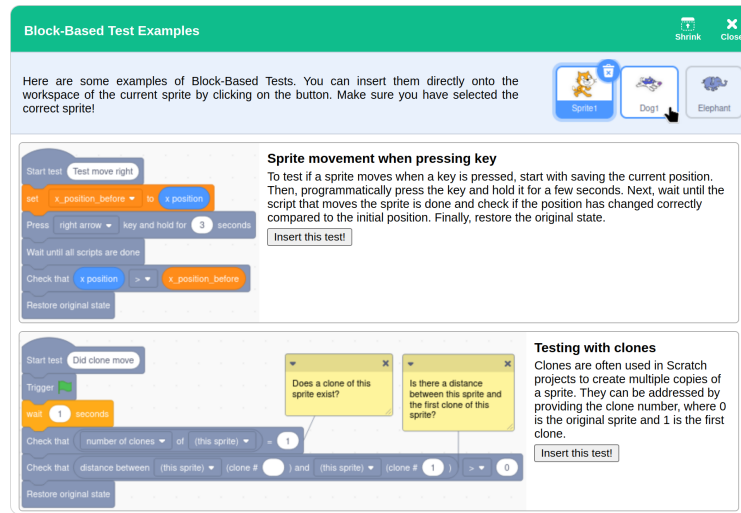
Fig. 5. The examples window showcases tests for common scenarios and allows users to import them into their project.

diverse set of reporter blocks shown in Fig. 2d to our testing framework. These blocks allow users to easily access a sprite's position, direction, costume number, costume name, size, volume, displayed saying/thinking text, sprite clones, and the number of currently running scripts. In addition, these blocks enable users to select a reference to the current sprite instead of a specific name, which simplifies the reuse of test scripts across different sprites. All reporter blocks are designed to be compatible with existing SCRATCH blocks and can also be used in regular program scripts.

4 CREATING AND EXECUTING TESTS

4.1 Creating Tests

From the user's perspective, block-based tests can be created similarly to regular scripts by either utilising the *drag-and-drop* functionality from the toolbox, duplicating existing tests on the workspace or copying tests across different sprites. Leveraging the extension support of the virtual machine, these test blocks are designed to integrate with the existing scope of functionality. Therefore, test blocks can be combined with regular SCRATCH blocks to offer a powerful toolset for its users. For instance, in order to check some condition within a test case, conditional blocks that are part of the SCRATCH *operator* category can be inserted into an assertion block that seamlessly evaluates the native condition. Just like in regular scripts, native blocks within test scripts can implicitly refer to the currently selected sprite. The reporter blocks introduced in this extension (Section 3.4) however have an option to explicitly select the concerning sprite.

Since writing tests may be challenging for programming beginners who have never tested programs before, our tool includes a dedicated examples window (shown in Fig. 5) to ease the introduction to block-based testing. This window grants users access to a curated selection of pre-constructed tests. These examples span a variety of common scenarios, such as testing the movement of sprites upon arrow key activation, and provide practical help for users to understand and implement testing strategies. They can be imported to the workspace by pressing a single button. This feature significantly lowers the barrier to creating tests by eliminating the need to develop them from the ground up, thereby making the first steps into testing more approachable and less intimidating for users. Since the template tests are inserted as editable scripts, users can customise them according to their needs.
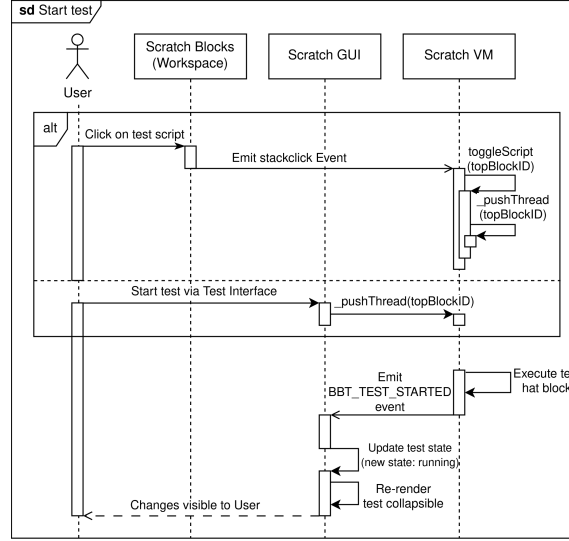
Fig. 6. Simplified sequence diagram showing the interaction of SCRATCH components when a test is started.

4.2 Test Execution

A test script can be initiated by either directly clicking on it in the workspace or by pressing the run button of an individual test (or the complete test suite) within the test interface (Section 5). As described in Fig. 6, the workspace issues a *stackclick* event when a (test) script is clicked, to which the virtual machine responds by executing the *toggleScript* function. This function then pushes a new thread, which manages the execution of the clicked test script, into the list of active threads. In contrast, if a single test or the entire suite is started through the test interface (Fig. 7b), the respective threads are directly pushed to the list of active threads. As soon as the activated thread of the test is executed by the sequencing function of the virtual machine, the *BBT_TEST_STARTED* event is fired in order to notify the user interface about the started test. The user interface reacts to this event by updating the tests' status in the test interface to *running* and clearing any errors from previous test executions.

In the ongoing execution process, test threads are treated like all other program-related threads and do not have special precedence during the execution, which might require the use of test-related control blocks in specific scenarios (Section 3.1). Whenever it is the turn of a test thread, the test is executed like any other program script by performing the actions of the respective blocks that are hosted by the script. In case one of the assertion blocks introduced in Section 3.3 is executed, the test interface is notified about the evaluation result, enabling it to update the visualisation of the test in both the workspace and the interface itself with a green or red colour for a passed or failed test, respectively. While tests are running, real user input and interactions with testing functionalities are temporarily disabled to avoid ambiguous testing behaviour.

A running test script can end due to one of three reasons. First, a test may conclude naturally after the test script has been fully executed, which causes the respective thread to be retired by the virtual machine. Secondly, a test can come to an early end if the user decides to abort the program execution by clicking on the native 🔴 button, as this causes the virtual machine to retire every active thread immediately. Finally, a test may be interrupted if it encounters a timeout that is set implicitly by the `Start test Test 1` block or explicitly by the `Set timeout to 60 seconds` block. In order to ensure

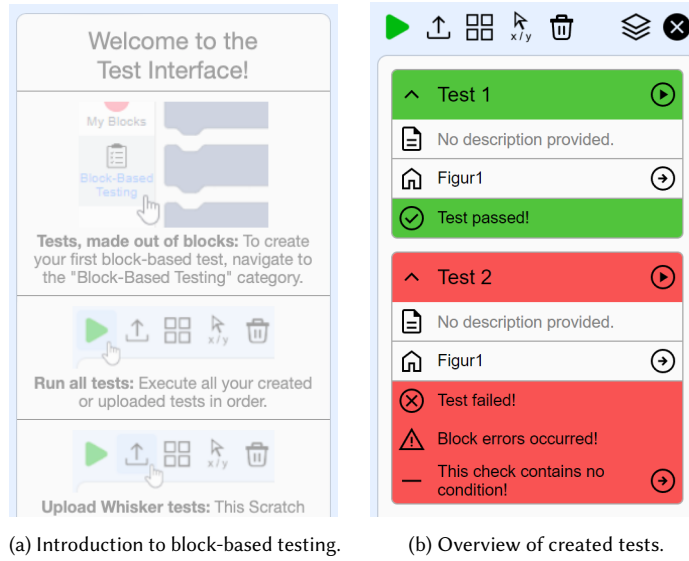(a) Introduction to block-based testing.   (b) Overview of created tests.

Fig. 7. The test interface shows an introduction to block-based testing if no tests exist, or an overview of all existing tests.
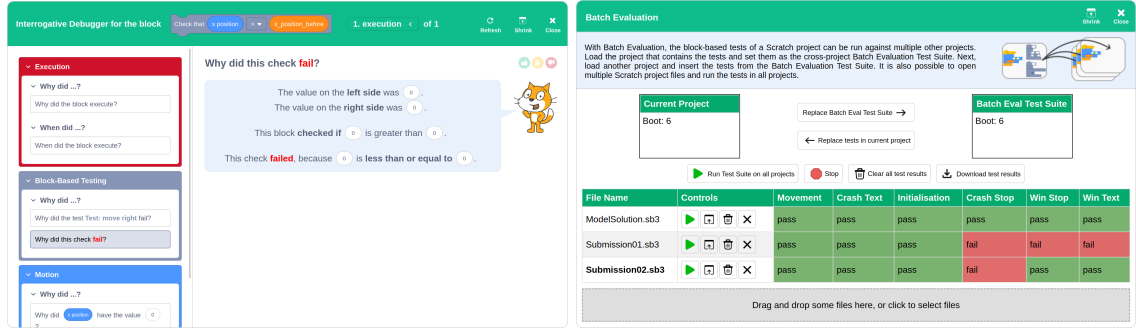
that successive test cases of a test suite are not impacted by changes in the program environment that may result from a partially executed test that ran into a timeout, the program state is reset using the snapshot of the `Start test Test1` block (Section 3.1), regardless of the presence of a `Restore original state` block. Furthermore, all mouse and keyboard inputs are reset to prevent lingering inputs from affecting successive test executions. Note that failing assertion blocks do not lead to early termination and test execution will continue regularly.

Following the completion of each test, irrespective of the manner of its conclusion, the control buttons of the test interface are re-enabled, and the processing of human user inputs resumes. If tests ended naturally or due to timeouts, the test execution logic contained in the interface will automatically initiate the next test in the sequence, provided the current test was started as part of a test suite. However, for tests that were aborted, advancing to the next test would counter the user's deliberate choice to cease all activities by clicking the stop button and is therefore avoided.

5 INTEGRATING BLOCK-BASED TESTS INTO THE SCRATCH ENVIRONMENT

The test interface depicted in Fig. 7 is the main control component to manage and observe block-based tests. It is integrated within the existing components of the graphical user interface between the workspace and the stage. A major focus was put on maintaining the editor's visual harmony by adhering to the design principles of Scratch to create an interface that appears both native and intuitive.

Initially, this area introduces users to the concept of block-based testing through a quick start guide (Fig. 7a), supporting those new to test creation. As soon as the first test is created, this section evolves to host collapsible block entries for each test script (Fig. 7b). These entries provide information about each test, including its name, description, execution controls and contextual details such as the associated sprite. Additionally, these entries display and highlight errors, using a red/green colour scheme for immediate visual feedback on test outcomes. To assist users in debugging, we combine the proposed block-based testing framework with the interrogative debugger NuzzleBug (Section 5.1).

(a) The interrogative debugger NuzzleBug is extended with a category to ask questions about the execution of tests.

(b) The batch evaluation window allows users to execute a test suite on multiple similar Scratch programs.

Fig. 8. Extensions integrating block-based testing into the Scratch environment.

The control bar, located at the top of the test interface (see Fig. 7b), offers a range of functionalities that cater to different testing needs, including running all tests sequentially ▶ (Section 4.2) and displaying the examples window ▦ introduced in Section 4.1. Using the ⬆ button, users can upload Whisker tests (Section 2), which are then integrated as separate test cases to the test interface like other block-based tests. Furthermore, we help users in writing tests that involve precise coordinates by offering a tool ⬏ that dynamically displays the current coordinates of the mouse pointer within the stage. The 🗑 button can be used to clear all test results and the ⬚ button gives users access to execute created tests on a batch of Scratch programs (Section 5.2). Finally, the ⊗ button collapses the test interface area to avoid cluttering the Scratch interface if users are not interested in writing tests.

5.1 Debugging Test Outcomes

Test results are visually indicated within the workspace by a glowing effect around the respective test script, as well as through entries within the test interface. If a test fails, the interface provides detailed information about the nature of the error, pinpointing the exact location where the issue occurred. Users can quickly navigate to the problematic block by clicking the ⊙ button next to the error entry in the test interface. To further assist users in debugging, we extended the interrogative debugger NuzzleBug [5] with reasoning about block-based tests, as shown in Fig. 8a. This integration allows users to pose questions about the reasons for failing tests and assertions. For instance, Fig. 8a involves an example where the debugger is queried about a failed numerical assertion block. The interface outlines the necessary conditions for the assertion block to succeed, thus offering a deeper insight and enabling users to fix their implementation.

5.2 Batch Testing of Scratch Programs

By clicking on the ⬚ button, users can launch the batch processing window shown in Fig. 8b, which allows them to execute a block-based test suite across multiple Scratch programs. This functionality is orchestrated through the concept of a cross-project *batch test suite*, wherein tests from one project can be designated as a suite to be applied across various others. Upon loading a project into this environment, users can inject tests from the *batch test suite* into the new project while replacing all existing tests. The content of the *batch test suite* is not bound to the originating project file, but can be dynamically updated or modified within any project into which it has been previously injected, thereby enhancing the flexibility and adaptability of the evaluation process and tailoring it to evolving testing requirements.

The lower section of the batch evaluation window is occupied by the program table, which presents a comprehensive listing of all SCRATCH programs currently loaded into the batch processing environment. New programs can be added to this table via *drag-and-drop* or by selecting SCRATCH files using the operating system's file browser. The second column of the table is equipped with several controls, enabling users to automatically execute all tests from the *batch test suite* in a specific project, load projects into the editor, remove file-specific test results or remove the complete file entry. Whenever the batch evaluation procedure is initiated, each project of the file table is first loaded into the SCRATCH instance and then evaluated against the *batch test suite*. The result of each test case is then recorded in the file table, where each test case is represented by a column named after the respective test case as defined by the ▢Start test Test 1 block.

6 EVALUATION

To understand how block-based testing helps teachers assess students, we consider the following research questions:

- **RQ1:** How accurately do the teachers' tests match the task description?
- **RQ2:** How accurate is the teachers' final assessment?
- **RQ3:** What do teachers think about block-based testing?

6.1 Experimental Setup

6.1.1 Experiment Participants. We conducted a pilot study with five in-training teachers without a computer science background, introduced them to SCRATCH and refined the study setup based on the insights. Data from the pilot study was not included in further analyses. The main study was conducted with 20 practising teachers participating in professional training and 8 in-training teachers. In the following, we will refer to all of them as teachers.

6.1.2 Experiment Procedure. Since only five teachers had used SCRATCH before and the others had little to no experience, we started with a two-hour introduction to SCRATCH, in which the teachers were tasked to implement smaller exercises, a larger game, and a code understanding task explaining the scripts of the *Boat Race* tutorial[1]. In a second session, we started with an introduction to block-based testing, including an exercise to write a test for a task from the introductory session. We included examples and exercises using all test blocks and strategies needed throughout the study and provided cheat sheets summarising the introduction. Following the introduction, the teachers received the *Boat Race* SCRATCH project and were tasked to write tests for it. Since writing tests without knowing specific details about the task would be unrealistic in a teaching scenario, we specified six functionalities to be tested based on the tutorial [16]:

- (a) The boat starts in the harbour with the costume "normal".
- (b) The boat follows the mouse pointer.
- (c) When the boat crashes into the wall, it says something.
- (d) After crashing into a wall, the boat's costume is "damaged".
- (e) When the boat reaches the beach, it says something.
- (f) After reaching the beach, the boat's costume is "festive".

After all participants felt they had invested sufficient time in creating their tests, we demonstrated batch testing (Section 5.2) and explained how to change and refine tests while inspecting student solutions. We then supplied the 28 teachers with 21 student solutions for the *Boat Race* program [16] and tasked them to assess the six functionalities listed above for at least five solutions in the remaining time of a total study duration of three hours. Although they were

---

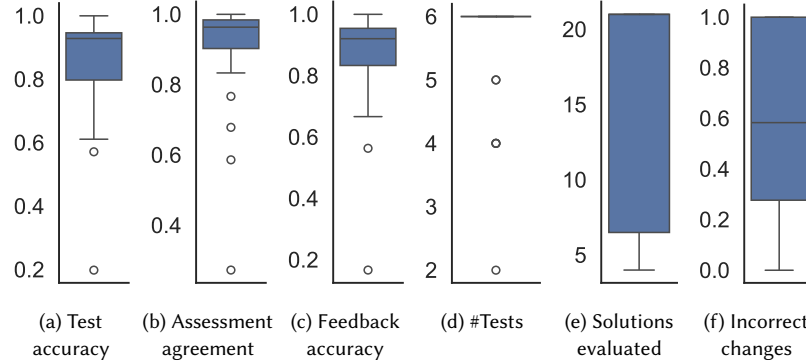[1]https://projects.raspberrypi.org/en/projects/boat-race

Fig. 9. The accuracies comparing teacher tests and gold tests, teacher tests and assessment and teacher assessment to the gold tests, the numbers of tests written and students assessed, and the share of changes to the test results not fitting the student.

told to use their test results, they were free to manually inspect the projects, change their test cases, or simply provide assessment differing from the test results. Assessment had to be submitted via an online form, where the teachers could tick whether the student implemented each functionality correctly or not, and add textual explanations to the given *correct/not correct* decisions. At the end of the experiment, the teachers were given a survey consisting of nine questions, asking their agreement on a 5-point Likert scale to questions regarding their experience with the framework, and three free-text questions about issues encountered, suggestions for improvement, and general remarks.

6.1.3 Analysis Procedure. **RQ1:** To determine the accuracy of the teachers' tests, we created a *golden test suite* consisting of 6 tests (one for each required functionality), which we manually refined to provide the correct outcomes for all student solutions. We then executed each teacher's test suite on the 21 student solutions, and calculated accuracy as the matching ratio of pass/fail verdicts between our golden test suite and the teachers' tests to the overall number of tested functionalities (21 solutions × 6 functionalities). If a teacher did not write a test for a functionality, we count this as a mismatch for each student solution. We answer RQ1 by reporting the distribution of accuracy values across all teachers. **RQ2:** To determine the accuracy of the teachers' assessment provided via the online form for each participant, we ascertain the ratio of *correct/incorrect* assessment results matching the *pass/fail* results of our *golden* tests on the subset of student solutions evaluated. We additionally consider the ratio of *correct/incorrect* assessment results matching the *pass/fail* results of *their own tests* to determine disagreement between test and assessment results. **RQ3:** In order to analyse the opinions of teachers on the block-based testing framework, we summarise the 5-point Likert scale answers and the free-text answers for each question of the exit survey.

6.1.4 Threats to Validity. Threats to *internal validity* may result from using a predefined task, as teachers tend to perform better when working on a task they created and understand well. Threats to *external validity* may stem from the number of participants and the use of only one task and data set, limiting how well results may generalise to other tasks and student solutions. To counter this threat, we support replication by providing all source code and materials online. Threats to *construct validity* can arise from our focus on evaluation correctness, as we do not analyse the time needed or effects on other assessment attributes, such as helpfulness and verbosity.

6.2 RQ1: How accurately do the teachers' tests match the task description?

Figure 9a plots the distribution of accuracy values for each of the 28 participants' tests compared to the golden test suite. The median accuracy of 0.93 confirms that the tests cover the six specified functionalities very well most of the time, and we conclude that teachers are able to write tests useful as a reliable measurement for automated assessment.

Although Fig. 9d shows that most teachers wrote the 6 tests required to cover all aspects of functionality, there are a few who did not (2 participants wrote only 5 tests, 3 wrote only 4, and one participant wrote only 2). To some degree, these missing tests contribute to the loss of overall accuracy in Fig. 9a (median accuracy 0.94 when ignoring missing tests). For four out of the six teachers who did not write all tests, those tests they did write were good (overall accuracies between 0.61 and 0.80). Given more time, they likely would have eventually produced the missing tests.

The worst performing participant (accuracy 0.20), visible as a clear outlier in Fig. 9a, only wrote tests for two functionalities, and even these two tests were not very well suited for assessing the targeted functionalities: One test checks whether the boat touches a brown wall immediately after the green flag is clicked (which should always fail) and the other test simply positions the mouse pointer without ever asserting anything (thus consistently passing).

The other outlier in Fig. 9a (accuracy 0.57) wrote five (mostly fitting) tests, but one test, intended to check whether the boat says something when colliding with the walls, actually checks if the word *apple* is longer than 50 characters, which is nonsensical and always results in a failed test. Likely, this teacher tried to check if the boat says anything but then simply forgot to replace the default values of the SCRATCH blocks.

Summary (RQ1) Most teachers can write tests for common tasks after a short introduction to block-based testing.

6.3 RQ2: How accurate is the teachers' final assessment?

While RQ1 confirmed the teachers' capability to write useful tests, we now examine whether they also delivered proper assessments in the end. Figure 9c indicates highly correct assessments with a median accuracy of 0.92. Thus, overall, the teachers provided correct assessment. Figure 9b rates the agreement between the teachers' assessment and the results of their tests at a median of 0.96. This demonstrates that, by and large, the teachers' final choices are in agreement with their tests. However, there is a minimal decrease in the accuracy compared to the test results (from 0.93 to 0.92), as the teachers sometimes disagreed with their own tests. All but four teachers made at least one change between test outcomes and their assessment, resulting in 150 cases where assessment differed from the test outcomes overall. Figure 9f illustrates that these changes slightly tend to make things worse rather than better (median 0.58): 77 cases changed correct test outcomes to incorrect assessment, and 73 cases with incorrect test outcomes were corrected.

Out of the 77 changes to a wrong assessment, a total of 45 changes were passing tests for which some teachers apparently decided to be more strict than their block-based tests and our golden test suite. For example, one teacher produced a test suite with perfect accuracy, but then changed four test results as, in their eyes, the student solution was not good enough and had room for improvement (e.g., the boat gets teleported to the mouse pointer immediately after the initialisation, so it is barely visible). Other examples of this are two teachers who changed the assessment for the mouse following functionality to *not correct* even though the test passed, because of a long waiting block before the boat starts its motion, which probably was not the reaction the teachers expected. Another teacher disagreed with the test result on the costume change to *damaged* when hitting the walls as *correct*, as the boat starts with the festive costume.

We also observed 82 cases where teachers overruled failing test results and decided the outcome was acceptable. In 50 cases, this fixed an incorrect test outcome, but in 32 cases, they accepted incorrect behaviour as correct. While this may be influenced by misunderstandings, there are probably also cases where teachers deliberately decided to be more

(a) Incorrect test.　　　　　(b) Fixed test, yielding explicitly.　　　　　(c) Alternative: implicitly yielding block.
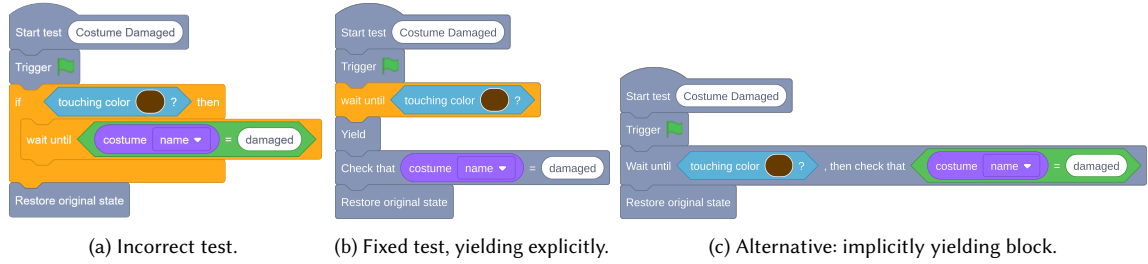
Fig. 10.   The test attempts to check if the boat costume changes after touching the colour of the wall; if this does not occur within 5 seconds, the test will time out. However, this check is performed only *once* after triggering the green flag, but the boat has not collided with the wall yet. Fixing the test includes waiting for the prerequisite condition and yielding afterwards.

lenient. Generally, in such cases of disagreement we would have liked teachers to engage more with refining their test suites to match their actual expectations, but it seems that in this aspect the framework may need future improvements.

Figure 9b shows the same outlier already discussed for RQ1, who only wrote two tests: One of the tests always *passes* as it does not check anything, resulting in disagreement with the assessment. Interestingly, no assessment was given on the functionality tested by the other test, but the teacher instead assessed a different aspect of functionality that was not tested at all. Given that this teacher assessed the fewest (only four) submissions (Fig. 9e), the problem was likely not just struggles with our framework, but general lack of engagement with the experiment or SCRATCH.

The other outlier discussed for RQ1 can also be seen as the second to lowest outlier in Fig. 9b, mainly affected by the test that always fails. This particular participant seemed to focus on giving good student assessment by also evaluating the functionality without tests. While this resulted in high accuracy of the assessment (0.87, Fig. 9c), the agreement between test results and assessment is comparatively low (0.59, Fig. 9b).

Another interesting outlier wrote six tests of which four always fail, even for correct student solutions, resulting for RQ1 in a low accuracy of 0.68. This is caused by tests performing a single collision check immediately after the green flag is activated (see Fig. 10), thus showing the common bug pattern *Missing Loop Sensing* [9]. An assessment accuracy of 0.9 for this participant shows that they were able to come to mostly correct conclusions despite their incorrect tests.

A further factor, contributing to the slightly lower accuracy of the assessment compared to the tests, are 12 cases in which teachers only provided an explanation for their assessment without actually selecting *correct/not correct*, with one even selecting both options. In our analysis procedure, this is also counted as a mismatch, as no definitive assessment was given. Such issues could be overcome by extending the batch testing interface to a full assessment tool, in which only test results deemed as incorrect need to be changed by hand.

Summary (RQ2) The teachers correctly assessed most functionalities, although they sometimes disagreed with their tests to be more strict or lenient without updating them.

6.4   RQ3: What do teachers think about block-based testing?

Figure 11 summarises the responses to the survey questions on a 5-point Likert scale and shows that the teachers are generally confident in using the test interface itself, although Fig. 11b suggests that there were some problems with the concept of block-based tests. The answers given for the free-text question *If there were problems: What were they?* mostly suggest that the teachers wanted more time to familiarise themselves with the tool, and that sometimes finding the right block was not as easy, which is a general problem with the SCRATCH interface. The teacher who had

(a) I am confident in using the block-based test interface.

(b) I had problems using block-based tests.

(c) I have not changed my tests once they have been written.

(d) The more student solutions I tested, the better my tests became.

(e) The block-based tests helped me to find my way around the student project more quickly.

(f) Block-based tests help me to see if the students have written program parts that work.

(g) To make it easier to give individual feedback, I would use block-based tests.

(h) I can use block-based tests that others have written.
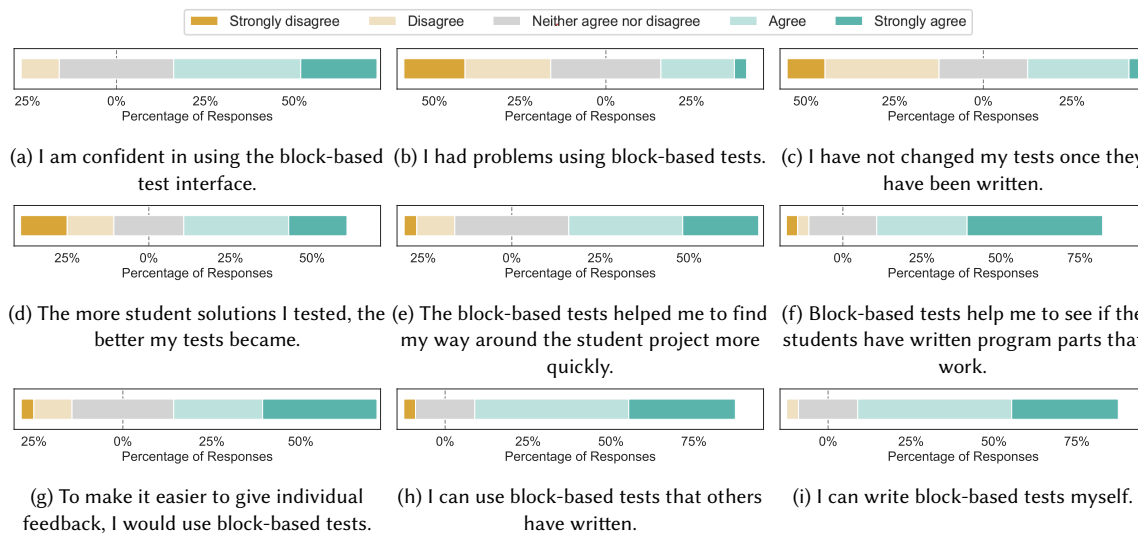
(i) I can write block-based tests myself.

Fig. 11. Survey results on how the teachers view block-based tests.

the lowest accuracy for RQ1 and RQ2 stated that they faced problems not knowing in which way they should test the functionalities. However, our introduction and the cheat sheet contained all necessary information about blocks and testing strategies. Maybe this is also an issue rooted in the limited time. The one teacher who strongly agreed to having problems with the tests had accuracies of 0.83 and higher in RQ1 and RQ2 and also strongly agreed to both being able to write tests themselves and being confident in using the interface, so this possibly was an incorrectly selected answer.

According to Fig. 11c, there is a slight tendency to change tests after the initial creation, and Fig. 11d also shows a slight tendency towards tests improving with more students being evaluated. However, the opinions seem to strongly diverge on this question. Future work could investigate the use of approaches like mutation analysis, simulating student mistakes [2] in order to guide teachers in producing adequate tests [3] prior to running them on student solutions.

While the majority of teachers consider block-based tests helpful to find their way around the student projects (Fig. 11e), a single *Strongly disagree* answer came from a teacher stating in the additional comments that they sometimes were confused by the order of their test cases in the interface and had a hard time looking up test results because of that. This teacher also suggested that a future version of the interface could include an option to reorder the tests. This perception likely negatively influenced the reported helpfulness of the tests. An even bigger majority (cf. Fig. 11f) agreed that block-based tests are helpful to check if specific parts of student projects are working. Most of the teachers also stated that they would use the block-based tests to simplify providing individual feedback (Fig. 11g). Again, the single *Strongly disagree* came from the teacher who had issues with the test order, which may explain why they did not find it easier, as they had problems looking up the test results.

Considering Fig. 11h and Fig. 11i, the vast majority of teachers think that they can write block-based tests themselves as well as use tests from other persons. However, in the latter category one teacher selected *Strongly disagree*, stating that they never used tests from other persons as additional comment. This may be a misunderstanding, as we intended to ask if they would be willing to use it with other people's tests, even if they would not be able to write tests themselves.

Summary (*RQ3*) The block-based test framework was generally well received, but met suggestions for improvement.

7 CONCLUSIONS

Extending SCRATCH with block-based tests supports teachers when assessing student solutions and providing feedback. Furthermore, they enable automated feedback and guidance, and an easy way to write tests may be a valuable approach for introducing learners to the concept of testing. In order to make this possible, we introduced a SCRATCH extension that provides test blocks and test controls, integrated directly into the SCRATCH user interface. Our initial study provides evidence that teachers can work with tests to assess students, and the teacher feedback is encouraging.

Having contributed to the foundations of testing in SCRATCH, there are now ample opportunities for future research:

- Our study revealed ideas for improving the interface itself, for example by adding a way to rearrange the displayed test order or to search for specific blocks in the SCRATCH toolbox.
- There is potential for adding further testing blocks. For example, inspired by Voeten's master thesis [22], the ability to execute blocks as if they were part of another sprite's workspace would offer a practical approach to initialising a testing environment. Such blocks could eliminate the need to create specific scenarios through block-based, simulated user input (or manual changes) by allowing direct control over other sprites.
- Recurring test patterns, such as verifying sprite movements, could be integrated as dedicated blocks of greater abstraction. The current test-related blocks already allow low-level checking of any possible program behaviour, but higher-level blocks for specific issues could improve the efficiency of creating tests. To achieve this, it will be essential to gather data on tests created by teachers to identify common patterns.
- While in this paper we focused on tests in which assertions check task-specific expected functionality, there may be general patterns of program misbehaviour or runtime errors that test executions can reveal, like sprites at the edge of the stage being unable to move further despite otherwise correct movement logic. Our testing framework provides the foundation for investigating such generalisable runtime checks to complement test assertions.
- The example test in Fig. 10 suggests that there can be common patterns of problems in block-based tests, akin to test smells [18, 21] in automated tests written in textual programming languages. Similar to how code smells can be defined and automatically detected for regular SCRATCH code [9], teachers and students may benefit from patterns and automated checks for block-based test smells.
- While our initial motivation is to support teachers, there is also potential to integrate block-based tests directly into education, leading to the question of how learners should work with tests. For example, learners could be introduced to a test-driven development approach [6] where they write tests prior to the code they intend to implement, or they could be taught to create automated tests to simplify debugging.
- Research on explaining test failures to support debugging will be important, for example by providing more elaborate textual explanations, suggesting debugging hypotheses, or providing fix suggestions.
- The test interface already provides an option to load and execute tests generated with the WHISKER automated test generation tool. It would be useful to extend automated test generation, as for example provided by WHISKER [4], to directly create block-based tests rather than using other programming languages as output.

In order to support future research, we provide a full replication package including all data and source code:

https://figshare.com/articles/dataset/25710288

A live instance of our extended SCRATCH version is available at:

https://scratch.fim.uni-passau.de/block-based-testing

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] Benjamin Clegg, Siobhán North, Phil McMinn, and Gordon Fraser. 2019. Simulating student mistakes to evaluate the fairness of automated grading. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 121–125.

[3] Benjamin S Clegg, Phil McMinn, and Gordon Fraser. 2020. The influence of test suite properties on automated grading of programming exercises. In *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 1–10.

[4] Adina Deiner, Patric Feldmeier, Gordon Fraser, Sebastian Schweikl, and Wengran Wang. 2023. Automated Test Generation for Scratch Programs. *Empirical Software Engineering* 28, 3 (2023). https://doi.org/10.1007/s10664-022-10255-x

[5] Adina Deiner and Gordon Fraser. 2024. NuzzleBug: Debugging Block-Based Programs in Scratch. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE'24)*. ACM, 13 pages. https://doi.org/10.1145/3597503.3623331

[6] Stephen H Edwards. 2003. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceedings of the international conference on education and information systems: technologies and applications EISTA*, Vol. 3. Citeseer.

[7] Benedikt Fein, Florian Obermüller, and Gordon Fraser. 2022. CATNIP: An Automated Hint Generation Tool for Scratch. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 124–130. https://doi.org/10.1145/3502718.3524820

[8] Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. 2021. Litterbox: A Linter for Scratch Programs. In *Proceedings of the International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET'21)*. IEEE, 183–188. https://doi.org/10.1109/ICSE-SEET52601.2021.00028

[9] 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

[10] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation (ICST'21)*. IEEE, 148–158. https://doi.org/10.1109/ICST49551.2021.00026

[11] 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

[12] 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.

[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. https://doi.org/10.1145/1868358.1868363

[14] 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)*. ACM, 103–109. https://doi.org/10.1145/3341525.3387365

[15] Herart Dominggus Nurue and Jeff Gray. 2024. A Testing Extension for Scratch. In *Proceedings of the 2024 ACM Southeast Conference* (Marietta, GA, USA) *(ACMSE '24)*. Association for Computing Machinery, New York, NY, USA, 266–271. https://doi.org/10.1145/3603287.3651217

[16] Florian Obermüller, Luisa Greifenstein, and Gordon Fraser. 2023. Effects of Automated Feedback in Scratch Programming Tutorials. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 396–402. https://doi.org/10.1145/3587102.3588803

[17] Florian Obermüller, Ute Heuer, and Gordon Fraser. 2021. Guiding Next-Step Hint Generation Using Automated Tests. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 220–226. https://doi.org/10.1145/3430665.3456344

[18] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2022. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering* 27, 7 (2022), 170.

[19] 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.

[20] 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.

[21] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 92–95.

[22] Iwijn Voeten. 2023. Een blokgebaseerd testframework voor Scratch. (2023). Master Thesis. http://lib.ugent.be/catalog/rug01:003150096. Scratch instance available at https://scratch.ugent.be/poke/editor/.