University of Passau

Chair of Software
Engineering II

Whisker: Automated Testing of Scratch Programs

Marvin Lorenz Kreis
MtrNr.: 77280

Prof. Dr. Gordon Fraser

In partial fulfillment of the requirements for the degree of
B.Sc. Computer Science

2019-02-21

Kreis, Marvin:
Whisker: Automated Testing of Scratch Programs
University of Passau, 2019

Contents

List of Figures, Tables and Listings

Abstract

Scratch is commonly used to introduce students to the principles of computer programming. As some courses in schools and universities are attended by a large number of students, one big disadvantage of Scratch becomes relevant: Grading Scratch assignments is troublesome. Scratch's interactive nature makes grading very time-consuming, since many key presses and clicks are required to execute a program. At the same time, automating this process is still an open problem, since Scratch's visual and auditory output make automation difficult.

In order to solve this problem, we implemented Whisker, a program which automates Scratch 3.0's input and output mechanisms to make functional testing for Scratch possible. Whisker offers a JavaScript interface, that allows users to simulate input events on programs, and to access Scratch's visual output in the form of sprite attributes and variables. Additionally, Whisker offers automated input generation, which can be used to control Scratch programs automatically. With this, we explore a property-based approach to testing by defining constraints, that the program under test must hold.

To evaluate Whisker, we tested a collection of student solutions from a sixth and seventh grade Scratch workshop. Our test results closely match scores from independent manual grading, with an average Pearson's correlation coefficient of 0.882 over ten test executions. Furthermore, we evaluated Whisker's automated input generation by measuring its statement coverage on the sample solutions to Code Club's Scratch courses, on which was able to achieve a mean statement coverage of 95.25% over ten runs.

Chapter 1

Introduction

1.1 Motivation

Introductory computer science (CS) courses often use educational programming languages to teach the principles of computer programming. These languages are designed to be easily understandable and engaging for programming novices. In order to accomplish this, they often feature visual and auditory projects instead of textual input and output (IO). One of the best known and most used languages for this purpose is MIT's Scratch [9, 6]. Usages of Scratch can be found from primary school classes all the way up to introductory CS courses in universities [4]. Scratch features a block-based programming environment, which lets users build interactive, multimedia programs with little effort.

But Scratch's multimedia focus can also be its downfall. Scratch is entirely developed and executed inside of a graphical user interface (GUI), and programs usually require manual interaction with keyboard and mouse to run. Because of this, grading student assignments is troublesome and particularly slow. Assessment of Scratch programs involves opening, running and interacting with every program individually. Some courses, that use Scratch, are attended by more than 200 students [4]. Therefore, manual assessment of student solutions becomes too time consuming to be feasible, and automated assessment is needed to grade assignments. For traditional programming languages with text-based IO, functional testing can be deployed in order to evaluate assignments in a less time consuming, less error-prone, automated way. Programs are given some pre-defined input, then their according output is analyzed and checked for correctness. But therein lies Scratch's problem. Scratch does not have textual input and output mechanisms like most programming languages. Therefore, automatically testing Scratch programs is not a trivial task and still poses an open problem.

Besides automated grading, dynamic testing for Scratch programs can also be useful for students. For one thing, teachers can provide their students with test suites, so they can run tests on their implementations themselves. By doing this,

students can receive valuable feedback about their programs. They can possibly identify and fix errors in their solutions before submitting them. In some cases, maybe even a form of test driven development (TDD) can be adopted. With TDD, students write their program bit by bit in order to incrementally satisfy an existing test suite. Likewise, dynamic testing is also helpful for self-study. Online tutorials for Scratch could include a test suite for learners to test their programs with. Learners could verify their solution, or receive feedback on possible errors in their solution.

1.2 Contributions

This thesis introduces a new approach towards dynamic testing of Scratch programs by automating Scratch's IO. The testing procedure involves a program, which simulates user input (e.g. mouse movement or key presses) and allows access to Scratch's visual output by providing information about the program's sprites and variables. We present a way to perform property-based testing [3] for Scratch, which opens up the possibility to use arbitrary sources of input for testing, including automatically generated input.

As the main contribution of this work, we introduce Whisker, an implementation of the aforementioned program. Whisker provides an interface to control the Scratch virtual machine through JavaScript methods. Whisker's source code is available at `https://github.com/marvinkreis/whisker`. With this implementation, we also explore the possibility of automated test input generation through a combination of random input and simple static analysis on the Scratch project.

We evaluate Whisker with realistic Scratch programs from different educational workshops and courses. The results provide the following insights:

(1) Test results can be accurate enough to aid in grading Scratch assignments. Tests are able closely match the results of manual assessment and show fairly consistent results over multiple test runs. We measured an average Pearson's correlation coefficient of $r = 0.882$ for the correlation between our test results and independent manual scores. And we observed a percentage of 4.52% of test-project combinations showing inconsistent test outcomes over ten test executions.

(2) A combination of random input generation and simple static analysis can be used to generate input, which covers a big portion of typical Scratch programs' functionality. We were able to achieve an average of 95.25% statement coverage on a variety of projects with automatically generated input, while running the projects without inputs only resulted in 47.14% statement coverage.

(3) The testing process does not interfere with the execution of the programs under test. Specifically, additional computations for testing do not slow down the execution of tested programs.

1.3 Outline

In Chapter 2, we first provide a small overview over the Scratch programming language (Section 2.1). Then we examine previous approaches towards automated testing of Scratch programs (Section 2.2) and explain some general challenges, which have to be overcome in order to perform automated testing for Scratch (Section 2.3).

We explain our approach towards testing Scratch programs (Section 3.1), and how we realized this approach with Whisker, in Chapter 3. We describe the environment, in which tests are executed (Section 3.2), how tests are written, and what functionality Whisker provides for testing (Section 3.3). Afterwards, we explain various challenges of this testing approach (Section 3.4).

In Chapter 4, we describe how Whisker can be used to perform property-based testing. We explain the general idea of this approach (Section 4.1) and the testing procedure we use to achieve this (Section 4.2).

Furthermore, we describe the implementation of Whisker in Chapter 5. After we explain the implementation environment (Section 5.1) and the general architecture (Section 5.2) of Whisker, we explain how the controlled execution of Scratch programs is implemented (Section 5.3). The following sections then each describe the implementation of one of Whisker's features. Finally, we explain how Whisker measures statement coverage (Section 5.8).

In Chapter 6, we perform an empirical evaluation of Whisker. In the beginning, we list the research questions and give an overview of the experiments we conducted. Afterwards, we describe the projects and test suites we use for the evaluation (Section 6.1), Then, in each of the following sections, we explain the experiments and the indicators we use to answer each research question and describe our results. Firstly, we analyze the accuracy of the test results from our test suites (Section 6.2) and the test suites' flakiness (Section 6.3), then we evaluate our algorithm for generating automated test input (Section 6.4). Finally we examine if programs under tests are somehow influenced by the testing process (Section 6.5). Afterwards, we discuss the results (Section 6.6), and list possible threats to validity (Section 6.7).

Finally, in Chapter 7 we describe how Whisker could be extended in the future, then we conclude in Chapter 8.

Chapter 2

Background

This chapter will give some background about how Scratch works and why testing Scratch programs is a difficult task. It will also highlight some previous approaches towards automated assessment of Scratch programs.

2.1 Scratch

Scratch [9] is a programming language developed by the MIT Media Lab. Its main goal is to offer an intuitive language suited for programming novices and children. Scratch implements a block-based code system, which eliminates the possibility of syntax errors. It also allows users to easily integrate graphics and audio into their programs. Code is run inside a special Scratch virtual machine. Therefore, Scratch is exclusively developed and run through Scratch's web interface, which can be seen in Figure 2.1.

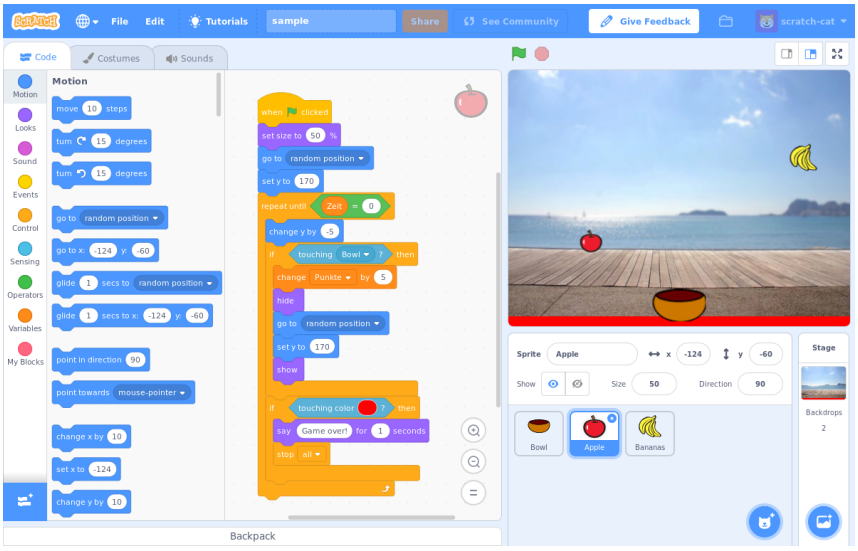


Figure 2.1: Scratch 3.0's web interface

Code. Programming in Scratch is done by sticking together a structure of pre-defined blocks, which are equivalent to statements and control structures in traditional programming languages. Multiple blocks are combined together with a *hat* to make up a *script*, as can be seen in Figure 2.2. Scripts are called through a type of event, which is defined by their hat. There are a variety of different events in Scratch. The main event is the *green flag* event, the entry point of the program. This event is emitted when the user starts the program by pressing the green flag button. The program then starts by executing all scripts that are equipped with a "when green flag is pressed" hat. Other events include key presses on the keyboard, clicks on a specific sprite, and events sent from different scripts. Active scripts effectively run in parallel until the end of each script is reached or until the script is stopped by itself or another script.

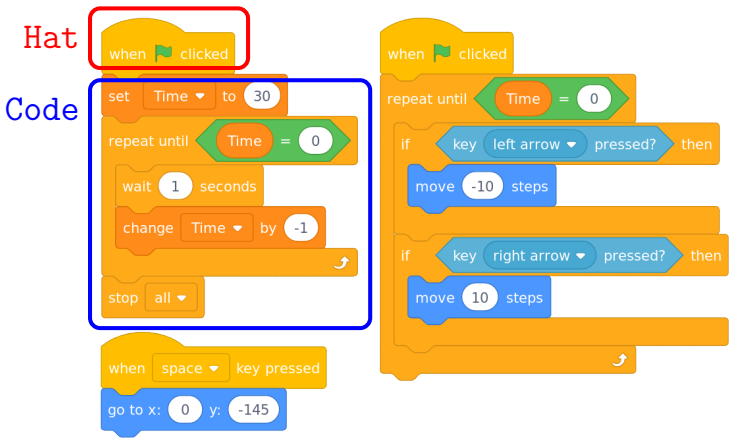
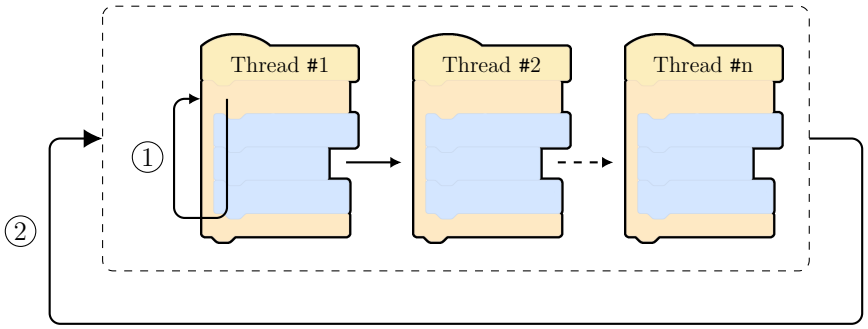


Figure 2.2: Scratch scripts

Input and output. Scratch programs create interactive two-dimensional animations through a number of visual objects (the *sprites*) on a background (the *stage*). The program manipulates sprites' appearance, position, size, rotation, visual effects and sounds. Furthermore, sprites can display messages and ask the user to type answers into a text box through `say` and `ask` blocks. Variables can also be displayed on screen. Programs can react to user input like keyboard key presses, mouse clicks or mouse cursor movement. Scratch's visual output is rendered in a constant frequency of 30 times per second. We will call the rendered pictures *frames*.

Sprite and variables. Each sprite contains its own scripts and variables. These variables, as well as the sprite itself can only be manipulated by the sprite's own scripts. The only exception to this rule is the stage, whose variables are global. Sprites can communicate with each other through a messaging system and through the stage's variables. By *cloning* sprites, the program can create multiple instances of a sprite, which behave alike. The cloned sprites then also share the original sprite's variables.

Program execution. Scratch executes part of the program in a fixed frequency of 30 times per second. We will call the program execution, that is done 30 times per second, a *step*. During the step, Scratch executes the program until either a certain time limit is reached, a sprite changes its position or appearance, or all scripts are done (and the program finished), whichever happens first. Then, a new frame is rendered. The program is executed by running its's threads, which each run an instance of an active script, sequentially. Each of the threads is executed until either the script's end is reached, a loop iteration has been executed, or the thread yields (e.g. when a `wait` block is used). Figure 2.3 illustrates this procedure.



(1) Execute until ...
- ... the end of script is reached or
- ... one loop iteration has been executed or
- ... the script yields

(2) Loop until ...
- ... a certain time limit is reached or
- ... a sprite's position or appearance changes or
- ... all scripts are done

Figure 2.3: Scratch step procedure

2.2 Previous Testing Approaches

Although automated assessment of Scratch programs is still an open problem, at least two other projects have tackled this task through different approaches.

Hairball. Boe et al. developed Hairball [2] to perform static analysis on Scratch programs. Hairball is implemented as a standalone Python program. It allows users to load a saved Scratch project file and analyze it by iterating through its blocks. An example application of this program is the web-based assessment tool called Dr. Scratch [7] by Moreno-León et al., which uses Hairball to measure the complexity of Scratch programs in various categories and scores them accordingly. Though Hairball is very useful for analyzing programs statically, testing the functionality of

a program with it would be very difficult. In order to do this, a dynamic testing approach is more suitable, because tests can simply observe to program's behaviour instead of having to deduct it from the programs source code.

ITCH. ITCH (Individual Testing of Computer Homework for Scratch Assignments) [4] by David E. Johnson is another Scratch assessment tool. Like Hairball, it is also implemented in Python. But it follows an entirely different approach. ITCH performs functional testing on Scratch programs, which have their functionality reduced to simple textual input and output operations. Scratch supports these operations through `ask` and `say` blocks. In order to automate the IO, ITCH replaces these blocks with structures, that give the program configured input text and record the resulting output. ITCH then executes the program, saves it, which also saves the current state of the variables, and analyzes the saved project file to generate a test report. This allows simple input-output-testing for Scratch, which is useful for testing the correctness of an implemented algorithm. But ITCH has a major drawback: Reducing Scratch programs to textual IO means that only a small subset of Scratch's functionality is available to the programs under test. Sprite manipulation and such cannot be tested with ITCH. If the only reason to adopt Scratch is its block-based code system, there exist other block-based programming environments, that serve as a front-end to more common programming languages, which can be tested more easily. An example for this is BlockPy [1], which translates its block-based code into Python.

2.3 Challenges of Testing Scratch Programs

Functional testing for Scratch is not a straightforward task. There exist multiple challenges, that have to be overcome in order to test Scratch programs. This section will go over some of these challenges and explain each of them.

Scratch's code system. Scratch does not have a traditional mechanism to structure code into methods, that return values. Unit tests often call methods with some input and check the returned value, but this is not possible in Scratch. Additionally, scripts run in parallel and may depend on one another. This makes it hard to test a part of the program in isolation from the rest of the program. One could circumvent this problem by restricting the tested programs, like ITCH [4] does it, but that defeats the purpose of using Scratch as a language.

GUI input and output. Scratch is entirely run inside a graphical user interface and lacks traditional IO mechanisms. Its output consists of visual animations and audio, which are difficult to analyze automatically. Likewise, Scratch's input, which mainly consists of keyboard and mouse input, can make interacting with the program in an automated way problematic.

Randomness. Scratch provides blocks to incorporate randomness into programs. Non-determinism is problematic for testing in general and not just for Scratch, but Scratch programs tend to frequently make use of randomness, since the programs implemented in Scratch are often of game-like nature.

Missing Initialization. Scratch programs don't start with a fixed configuration of sprites and variables. When the program is loaded, sprite attributes and variable values are restored from the project file, but subsequent program executions after the first one start with the configuration in which the last execution left the sprites and variables in. Scratch programs can deal with this by doing initialization in the beginning, but many implementations don't.

Fuzziness of properties. Scratch programs don't usually deal with exact values, since small differences in sprite properties are generally indistinguishable in the program's graphical output. This can make testing more difficult, since assertions may have to take small deviations into consideration. Scratch itself also makes dealing with exact values unattractive. In the Scratch GUI, sprites can initially be positioned on the stage via drag and drop, and instead of comparing exact positions, Scratch offers built in blocks to directly check for sprite collisions.

Tracking of properties. In order to test sprite movements and animations, sprite attributes need to be tracked over a period of time. Hence, sprite attributes need to be accessible periodically during the program execution.

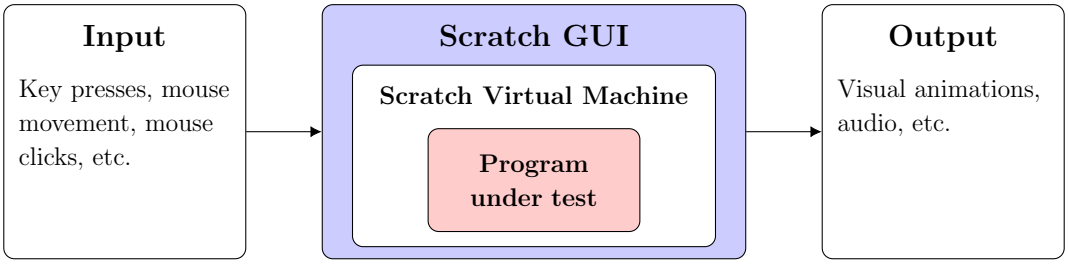
Chapter 3

Testing with Whisker

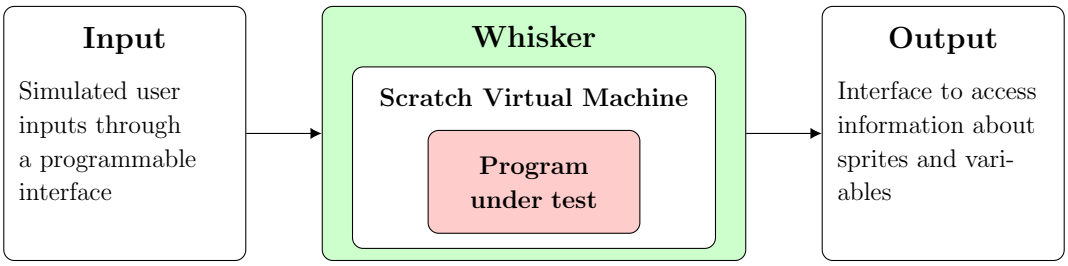
3.1 General Approach

In this work, we propose a way to perform dynamic testing on Scratch programs for Scratch 3.0. The main goal of this approach is to be able to automatically assess student's solutions to Scratch assignments. In order to do so, this approach makes use of an automation utility, which allows test code to interact with Scratch programs through Scratch's IO.

Because Scratch's parallel scripts, as well as its lack of code separation, would make testing individual parts of programs difficult, we instead chose to test full programs on the level of system tests, and only focus on the program's input and output. This raises the question of how to access Scratch's IO. Since its input usually consists of mouse and keyboard input, and its output consists of visual animations and sound, the IO is not easily accessible in a programmable way. To overcome this challenge, we developed an automation utility called Whisker, which acts as a wrapper around Scratch. It interacts with Scratch's virtual machine in order to automate its input and output. Whisker offers a programmable interface for Scratch, which allows tests to simulate inputs and to access information about sprites and variables. This makes automated testing for Scratch possible. Figure 3.1 illustrates the difference between Scratch's IO mechanisms and Whisker's automated input and output. Note that the current version of Whisker does not yet support audio output or Scratch's extensions.



(a) Input and output of the Scratch GUI



(b) Input and output of Whisker

Figure 3.1: Comparison of IO mechanisms between the Scratch GUI and Whisker

3.2 Testing Environment

Whisker is, like Scratch 3.0, implemented in JavaScript (JS). Hence, test code is also written in JavaScript. Whisker can theoretically be used with any JS testing framework, but for compatibility reasons, we developed a simple testing framework to go along with Whisker.

Currently, Whisker is only available in its own web GUI, which can be seen in Figure 3.2. The web page displays Scratch's stage, a table of loaded tests, and a test report in TAP13[1] format. It supports batch testing more than one program with the same test suite, but it doesn't support parallel test execution. In the future, we plan on implementing a standalone Electron[2] application for Whisker. This would facilitate batch testing many programs, and would also make it possible to test programs in parallel. Unfortunately it is not possible to run tests in a headless environment, because Scratch depends on a HTML canvas to render its output to. Without a renderer, some of Scratch's blocks don't work properly.

[1] Test Anything Protocol, `https://testanything.org/`
[2] `https://electronjs.org/`

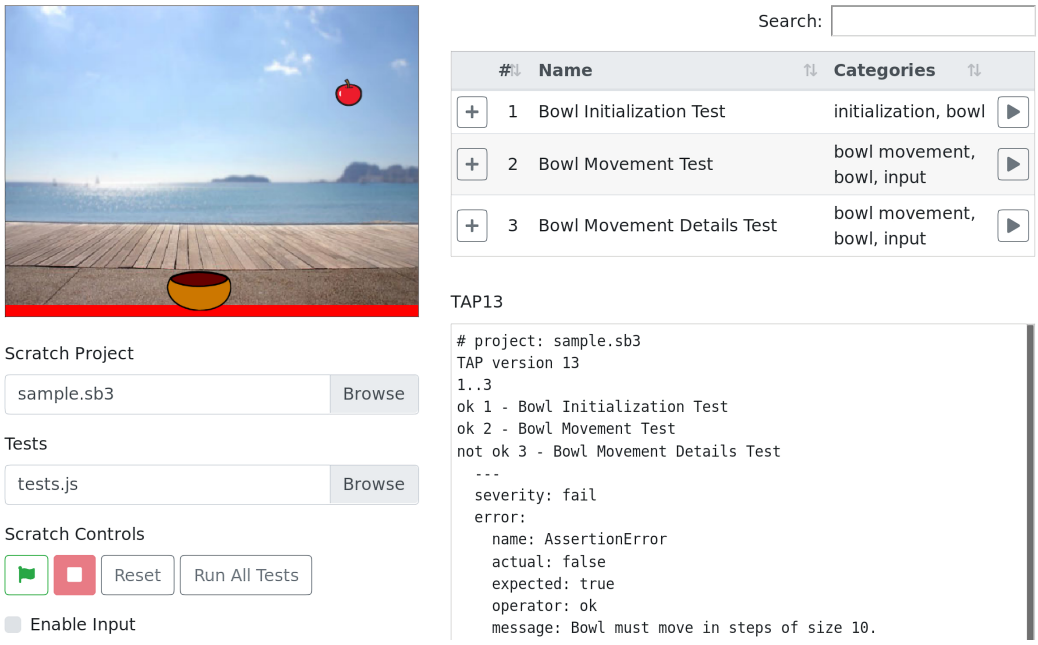


Figure 3.2: Whisker's web interface

3.3 Whisker's Public Interface

Tests use a test driver object to automate Scratch through Whisker. Whisker's own testing framework automatically passes the test driver object to its tests as an argument, but tests written for other testing frameworks may have to acquire the test driver in their test code. Whisker offers an interface to create and configure the test driver object for this purpose. Listing 3.3 shows code examples for both possibilities. Whisker loads the program before each test case in order to create the same initial state for every test, instead of going on where the last execution left off, like Scratch's GUI does it. It also starts the program with the green flag when the test begins.

```
1 async function test (t) {
2     ...
3 }
```

(a) Getting the test driver passed as a parameter

```
1 async function test () {
2     const whisker = new WhiskerUtil(vm, project);
3     whisker.prepare();
4     const t = whisker.getTestDriver();
5     whisker.start();
6     ...
7 }
```

(b) Manually acquiring the test driver through Whisker's interface

Listing 3.3: Acquiring the test driver

The following list will give an overview of Whisker's basic functions. Because Whisker is still in early development, some of its methods may undergo changes later. The test driver object will be denoted as t in example code snippets.

- **Control the program execution.** Tests are able to control when and for how long the program under test is run. In the beginning of the test, the program starts in a paused state. The program can then be run (resumed) for a certain amount of time, or until a condition is met. Since the Scratch program execution is asynchronous, we use JavaScript's Promise API for this purpose. In order to simply execute the program and wait until the execution is done, the test can be declared as an `async function` and use the `await` keyword.

  The green flag can also be pressed again in order to restart the program. Additionally, tests can query the currently elapsed time, either from the start of the last program execution, or from the start of the test case.

```
1  /* Run the program for one second (1000 ms or 30 steps). */
2  await t.runForTime(1000);
3  await t.runForSteps(30);
4
5  /* Run the program until a condition is met, or until one second elapsed.
6   * The condition is given by a function that returns a boolean. */
7  await t.runUntil(() => a > b, 1000);
8
9  /* Get the total time elapsed in the current / last run. */
10 t.getRunTimeElapsed();
11
12 /* Get the total time elapsed in this test case. */
13 t.getTotalTimeElapsed();
14
15 /* Press the green flag again. */
16 t.greenFlag();
```

- **Simulate Inputs.** By simulating Scratch's main input methods, tests can control the program under test. The goal is to simulate a user interacting with the program. The possible input includes mouse movement, mouse button presses, keyboard key presses, and entering answers to `ask` blocks. Inputs, that press (or release) a key or button, can specify a duration, after which the key is released (or pressed) again. Inputs can be performed immediately, or registered to be performed after a delay during the next execution. Inputs, that aren't performed during a run, will still trigger hats that respond to the input. Tests can also query the current mouse position and if a certain key or button is pressed.

  Note that inputs, that press a key for a long duration, do not issue multiple key press events. Usually, when holding down a key, the operating system repeats that key press, which will trigger "when key is pressed" hats repeatedly. This is not the case for simulated input. However, keys can be simulated to be

released and instantly pressed again, which will trigger "when key is pressed" hats.

```
/* Perform a keyboard input immediately. */
t.inputImmediate({
    device: 'keyboard',
    key: 'right arrow',
    isDown: true,
    duration: 100 // time in ms
});

/* Perform a mouse input one second into the next run. */
t.addInput(1000, { device: 'mouse', x: 100, y: 200, isDown: true });

/* Answer an ask block two seconds into the next run. */
t.addInput(2000, { device: 'text', text: 'some answer' });

/* Query the current state of the inputs. */
t.getMousePos(); // {x, y}
t.isMouseDown();
t.isKeyDown('space');
```

- **Access Scratch's output.** Whisker provides access sprites and variables of the program. They are accessed through sprite and variable objects, which always have the latest attributes of the sprite or variable they reference. Sprite objects can be obtained by their name or by their attributes. The provided sprite attributes include the sprite's position, rotation, size, current costume, speech bubble text, etc. Sprites also include the attribute values from the previous Scratch execution step, which can be accessed through the `old` attribute of the sprite objects. Additionally, sprites offer methods for detecting collisions, and other helper methods.

```
/* Get a sprite by its name. */
const sprite = t.getSprite('Sprite1');
/* Get sprites that match a condition. */
const sprites = t.getSprites(sprite => sprite.x > 100);
/* Get a sprite's clones. */
const clones = sprite.getClones();
/* Get the stage. */
const stage = t.getStage();

/* Various getter methods for variables. */
const variable = stage.getVariable('my variable');
const variables = stage.getVariables();
const list = sprite.getList('my list');
const lists = sprite.getLists();

```

```
/* Accessing sprite attributes and variable values.
 * Sprites and variables always have the latest values.
 * sprite.old has the values from the previous execution step. */
sprite.x;
sprite.old.x;
variable.value;

/* Some of the helper methods. */
sprite.isOriginal();
sprite.isTouchingEdge();
sprite.isTouchingSprite(otherSprite);
```

- **Register Callbacks.** Tests can execute code during the program execution by registering callbacks, which get called every time Scratch executes a step. Callbacks can be registered to be executed before or after each step execution. This allows tests to react to information, which the user would normally see, while the program is running. Callbacks can, for example, be used to track information, to perform inputs according to sprite information, or to cancel a run. Callbacks can be enabled and disabled.

```
/* Add a callback to be called before each step. */
const callback = t.addCallback(() => {
    if (sprite.x > 100) {
        t.inputImmediate({ device: 'mouse', isDown: true });
    } else if (sprite.x < 0) {
        t.cancelRun();
    }
});

/* Add a callback to be called after each step (true as 2nd parameter). */
t.addCallback(() => someList.push(sprite.x), true);

/* Enable / disable a callback, check if a callback is active. */
callback.disable();
callback.enable();
callback.isActive();
```

- **React to sprite movements and visual changes.** In addition to callbacks, which get called before or after each Scratch program step, tests can also set a function to be called whenever a sprite moves, or whenever a sprite's visuals change. The function takes the changed sprite as a parameter. This can be useful for detecting events, that happen during the step, but are not displayed to the user, which is needed for some edge cases. For example, consider program with two scripts: One of the scripts moves a sprite around. The other script moves the sprite back to the center of the screen whenever it touches on of the stage's edges. Therefore, whenever the sprite touches an

edge, it is immediately moved back to the center, even before the image is rendered. A normal callback will never detect the sprite touching an edge, but this way, detecting these collisions is possible.

```
let touchedEdge = false;

t.onSpriteMoved(sprite => {
    touchedEdge = touchedEdge || sprite.isTouchingEdge();
});

t.onSpriteVisualChange(sprite => { /* ... */ });
```

- **Register constraints.** By registering constraints, tests can define conditions that must always hold. Constraints are realized through special callbacks, which perform one or more assertions. Whisker can be configured to fail the test on a constraint failure (the default) or to simply disable the failed constraint. In the second case, it is up to the test code to check which constraints failed and react accordingly.

```
/* Configure the action taken when a constraint fails. */
t.onConstraintFailure('fail');
t.onConstraintFailure('nothing');

/* Register a constraint to check if some sprite is always visible. */
const constraint = t.addConstraint(() => {
    t.assert.ok(sprite.visible === true, 'Sprite must always be visible.');
});

/* Enable / disable a constraint, check if a constraint is active. */
constraint.disable();
constraint.enable();
constraint.isActive();
```

- **Perform randomly generated inputs.** Whisker provides random input generation by performing a randomly chosen input from a pool of possible inputs in a specified interval during program executions. Tests can either provide the pool or inputs to choose from themselves or they can let Whisker determine what inputs the program reacts to through static analysis. If a input consists of multiple events (i.e. if it has a duration) the same input cannot be chosen again while still active. If all of the inputs in the pool are active, no random inputs are performed until at least one input is done.

  Inputs in the pool can specify intervals for their attributes, from which a value is randomly chosen when the input is performed. They can also be assigned a weight to make them more likely or less likely to be chosen.

```
/* Set the interval. A random input is performed every 150ms
 * during program execution. */
t.setRandomInputInterval(150);

/* Register random inputs for the pool manually. */
t.registerRandomInputs([
      { device: 'keyboard', key: 'left arrow', duration: [50, 100] },
      { device: 'keyboard', key: 'right arrow', duration: [50, 100] },
      { device: 'mouse', x: [-100, 100], y: [-100, 100], weight: 0.5 }
]);

/* Let Whisker detect inputs for the pool. */
t.detectRandomInputs({ duration: [50, 100] });
```

3.4 Challenges of this Approach

This section highlights various challenges that one can face when using this approach to aid in grading Scratch assignments.

General testing challenges. Firstly, automated testing in general introduces some challenges. For one thing, programs have to be well specified, since testing relies on the program's specification. Therefore, if the specification is too vague, testing can become difficult. Tests for imprecise specifications potentially need to consider more possible cases of how the program could behave. Likewise, conflicting interpretations of the specification between the test and the program may result in false negative test outcomes. Also, since the program is only tested as a whole, testing a single part of the program can be difficult. If some feature of the program under test depends on another feature being implemented correctly, but the latter does not work, the first feature can not be tested properly.

Addressing sprites and variables. In order to access information about sprites and variables, they need to be addressed in some way. Sprites can be addressed by their name or by any of their attributes, for example by their position. Nevertheless, if a sprite, that is needed for the test, can not be found, the test can not work properly. This can be a problem if a Scratch program deviates from the specification. For example, if students are given a template with sprite names already in place, some students could rename the sprites, causing tests to fail. However, errors like this can easily be detected through test reports.

Missing initialization. Another challenge are programs with missing initialization which are saved in a bad state. When the green flag is pressed, the Scratch GUI simply picks up where the last execution left off. Therefore, students might not

initialize their program properly, which can cause it to sometimes behave incorrectly when the green flag is pressed. A program with missing initialization may be saved in a bad state, meaning the program will behave incorrectly the next time it is started. Since we restore the same state in the beginning of each test case, such a program will always behave incorrectly during the tests although the implementation might be mostly correct.

Parallel initialization. Since Scratch runs all "when green flag is pressed" scripts in parallel, programs tend to mix their initialization with actual program execution. This may make it difficult to test initialization, since some initialized values may already change while others have not been initialized yet, meaning that the program is never in a clean initialized state.

Time-displaced events. Programs may have events that are supposed to trigger certain actions. For example, a sprite touching another sprite may cause a variable to be incremented. But the exact timing for the triggered action is often unknown, because the event is detected through a loop. Therefore, tests have to check if the action happens in a time interval after the event occurred. This may be done by tracking timestamps of the event and the action, and comparing these timestamps, but this can be expensive to do.

Chapter 4

Property-based Testing with Constraints

This section will describe why separating control of the program under test from the test code can be beneficial, and how this can be achieved by defining constraints, that are checked in the background.

4.1 Constraint Tests

Usually, tests will provide the program under test with inputs and check the resulting outputs. However, in many cases, a different approach is possible as well. Tests may use other sources of input and simply observe if the program's output is correct for the input provided by the source. QuickCheck [3] by Claessen et al., for example, uses this principle, to test the correctness of Haskell programs. In order to do so, tests define conditions, which the program must comply with. QuickCheck then automatically generates input for the program and checks if the defined conditions hold. Scratch programs can often be tested in a similar way. But in order to be able to do this, tests have to be made independent of the simulated input on the program. This will not just enable us to test with generated input, but with other input sources as well. For example, the program could be manually controlled by a person, or input could be recorded and played back. Whisker also offers its own implementation of automated input generation.

Using constraints allows us to define conditions, which the program must hold. Constraints check the program's compliance to the conditions by continually performing assertions throughout the program execution. Since this is done entirely in the background while the program under test is running, we can define constraints and just let the program run for some time. This way, it does not matter in what state the program is during its execution, or what inputs it is receiving. If a condition does not hold, the respective constraint fails.

4.2 Testing Procedure

Figure 4.1 shows a testing procedure, which uses the aforementioned approach. We will go through each step of the procedure with the aid of an example. Consider a program with a single sprite, which is supposed to move to the right when, and only when, the right arrow key is pressed. We want to write a test to check if the sprite's movement works correctly. For the sake of simplicity we are going to ignore the case of the sprite not being able to move right when touching the right edge of the screen.

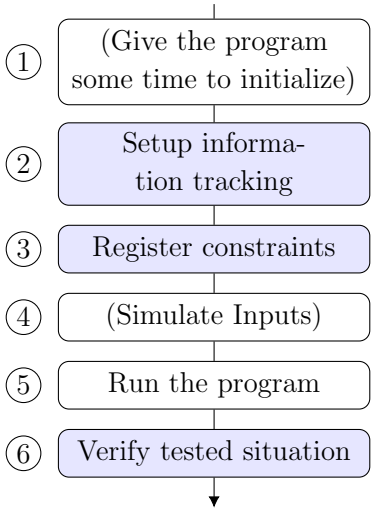


Figure 4.1: Constraint testing procedure

(1) **Give the program some time to initialize.** Before setting up any information tracking or registering constraints we may need to run the program for a short time. Otherwise the initialization of the program might be tracked, which will lead to wrong test results.

```
1 t.runForTime(100);
```

(2) **Setup information tracking.** Since the sprite might not move right in the same execution step the right arrow key is pressed, we want to accept slightly delayed movement in the test. For this purpose, we track the timestamps when the key starts being pressed, when the key was last pressed, and when the sprite last moved. We also record if the right arrow key was pressed (and released) at all, so we can verify it being pressed (and released) in the end of the test.

```
1 /* When the last right arrow key press started. */
2 let startPressedTime = undefined;
```

```
/* When the right arrow key was last recorded being pressed. */
let pressedTime = undefined;
/* When the sprite was last recorded moving right. */
let movedRightTime = undefined;
/* If the right arrow key was pressed in the previous step. */
let previouslyPressed = t.isKeyDown('right arrow');
/* If the right arrow key was pressed (and released) at all. */
let pressed = false, released = false;

const trackRightKeyCb = t.addCallback(() => {
    const currentTime = t.getTotalTimeElapsed();
    if (t.isKeyDown('right arrow')) {
        pressedTime = currentTime;
        if (!previouslyPressed) {
            startPressedTime = currentTime;
        }
        previouslyPressed = true;
        pressed = true;
    } else {
        previouslyPressed = false;
        released = true;
    }
});

const trackSpriteMoveCb = t.addCallback(() => {
    if (sprite.x > sprite.old.x) {
        movedRightTime = t.getTotalTimeElapsed();
    }
});
```

(3) **Register constraints.** Since we tracked when the time when the key was pressed and when the sprite last moved, we add a constraint that simply compares the timestamps to determine if the sprite moved at the correct times. We chose to allow a delay of 100 ms for the sprite to react to the right arrow key being pressed or being released. We also register a second constraint, which makes sure that the sprite only ever stays still or moves to the right, but never moves in any other direction.

```
const moveTimingsConstraint = t.addConstraint(() => {
    const currentTime = t.getTotalTimeElapsed();
    if (currentTime > startPressedTime + 100) {
        t.assert.ok(Math.abs(movedRightTime - pressedTime) <= 100,
            'Sprite must move right when, and only when, the right arrow
            ↪  key is pressed.');
    }
});

const moveDirectionConstraint = t.addConstraint(() => {
```

```
10      t.assert.ok(sprite.x >= sprite.old.x,
11          'Sprite must only stand still or move right');
12      t.assert.ok(sprite.y == sprite.old.y,
13          'Sprite must not move vertically.');
14  });
```

(4,5) **Simulate Inputs, Run the program.** Now we may register inputs and run the program. How we simulate inputs, or if we perform inputs manually, does not matter for the test, of course.

```
1  t.setRandomInputInterval(250);
2  t.detectRandomInputs();
3
4  await t.runForTime(1000);
```

(6) **Verify tested situation.** If our source of input does not guarantee that the right arrow key gets pressed at all, we risk the chance of a false positive test result, since the test never expects the sprite to move right. A similar problem occurs when the key is never released. Therefore, we want to handle cases in which the right arrow key was either never pressed, or was pressed the whole time. We may skip the test here, or just disable the constraint and mark it as skipped.

```
1  t.assume.ok(pressed, 'Right arrow key must be pressed.');
2  t.assume.ok(released, 'Right arrow key must be released.');
```

Figure 4.2 shows the resulting test code for the example and compares it to a similar test, which simulates inputs deliberately to check the sprite's movement. We can easily see that the constraint approach takes quite a bit more code. But at the same time, the constraint testing procedure is able to scale better. Once enough tracking is set up, writing constraints becomes easy and requires little code. At the same time, because the constraints are isolated from the program execution itself, many constraints can possibly be combined into a single test. For this purpose, Whisker offers an option to disable failed constraints instead of failing the entire test. In this case, tests can simply check which constraints failed, and let Whisker generate a test report from them.

```
const sprite = t.getSprite('Sprite1');

/* Give the program some time to initialize. */
await t.runForTime(100);

let oldX = sprite.x;

/* Check that the sprite never moves into an other
 * direction that right. */
t.addConstraint(() => {
    t.assert.ok(sprite.x >= sprite.old.x,
        'Sprite must only stand still or move
        ↪ right');
});

/* Run without inputs and check that the sprite
 * doesn't move. */

await t.runForTime(1000);

t.assert.ok(oldX === sprite.x,
    'Sprite must not move right when no key is
    ↪ pressed.');

/* Run with the right arrow key being pressed and
 * check that the sprite moves to the right. */

t.inputImmediate({
    device: 'keyboard',
    key: 'right arrow',
    isDown: true
});

await t.runForTime(1000);

t.assert.ok(oldX < sprite.x,
    'Sprite must move right when right arrow key is
    ↪ pressed.');
```

(a) Normal test

```
let sprite = t.getSprite('Sprite1');

/* (1) Give the program some time to initialize. */
await t.runForTime(100);

/* When the last right arrow key press started. */
let startPressedTime = undefined;
/* When the right arrow key was last recorded being pressed. */
let pressedTime = undefined;
/* When the sprite was last recorded moving right. */
let movedRightTime = undefined;
/* If the right arrow key was pressed in the previous step. */
let previouslyPressed = t.isKeyDown('right arrow');
/* If the right arrow key was pressed (and released) at all. */
let pressed = false, released = false;

/* (2) Track when the right arrow key is being pressed,
 * and when the sprite is moving to the right. */

const trackRightKeyCb = t.addCallback(() => {
    const currentTime = t.getTotalTimeElapsed();
    if (t.isKeyDown('right arrow')) {
        pressedTime = currentTime;
        if (!previouslyPressed) {
            startPressedTime = currentTime;
        }
        previouslyPressed = true;
        pressed = true;
    } else {
        previouslyPressed = false;
        released = true;
    }
});

const trackSpriteMoveCb = t.addCallback(() => {
    if (sprite.x > sprite.old.x) {
        movedRightTime = t.getTotalTimeElapsed();
    }
});

/* (3) Check if the sprite only moves when the right arrow
 * key was pressed, and if it doesn't move when the key was
 * not pressed. */

const moveTimingsConstraint = t.addConstraint(() => {
    const currentTime = t.getTotalTimeElapsed();
    if (currentTime > startPressedTime + 100) {
        t.assert.ok(Math.abs(movedRightTime - pressedTime) <= 100,
            'Sprite must move right when, and only when, the right
            ↪ arrow key is pressed.');
    }
}));

const moveDirectionConstraint = t.addConstraint(() => {
    t.assert.ok(sprite.x >= sprite.old.x,
        'Sprite must only stand still or move right');
    t.assert.ok(sprite.y == sprite.old.y,
        'Sprite must not move vertically.');
}));

/* (4) Some code, which registers inputs. Or nothing if
 * inputs are done manually. For example, generated input: */
t.setRandomInputInterval(250);
t.detectRandomInputs();

/* (5) Run the program. */
await t.runForTime(5000);

/* (6) Check if the right arrow key was pressed at all,
 * and if it was ever released. */
t.assume.ok(pressed, 'Right arrow key must be pressed.');
t.assume.ok(released, 'Right arrow key must be released.');
```

(b) Constraint test

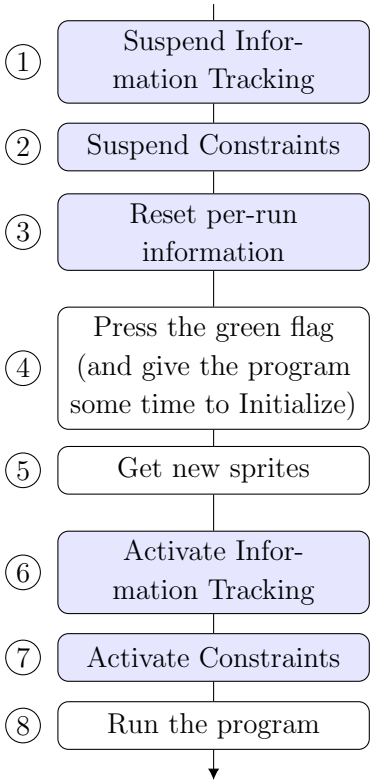
Listing 4.2: Comparison of a normal test and a similar constraint test

4.3 Resetting the Program During Tests

When testing with randomly generated input, programs often need to be reset multiple times inside of one test execution, because the test could get stuck in some state of the program.

To reset a Scratch program, we may simply press the green flag button again, but we also need to consider the tracked information as well as the registered callbacks and constraints when resetting the program under test. Any per-run information we tracked has to be cleared, and callbacks as well as constraints have to disabled before resetting the program. After resetting the program, callbacks and constraints have to be enabled again. Also, if Whisker was configured not to fail the test when a constraint fails, then the test needs to check which constraints failed before disabling them and then only enable constraints which were active before. Figure 4.3a shows a procedure that may be used to reset the program under test. This procedure replaces step 5 (Run the program) in the constraint testing procedure (see Figure 4.1). Additionally the code example in Listing 4.3b implements this procedure for the example from the previous section.

Sometimes, information may need to be tracked across resets in order to determine what events occurred in the program during the whole test execution. This information may be needed to rule out constraints, whose situation never occurred, after the test execution. An example for this are the variables `pressed` and `released` in Listing 4.2b.



(a) Procedure for resetting the program under test

```
for (let i = 0; i < 5; i++) {
    const constraints = [moveTimingsConstraint,
        moveDirectionConstraint];

    /* (1) Suspend information tracking. */
    trackRightKeyCb.disable();
    trackSpriteMoveCb.disable();

    /* (2) Suspend constraints.
     * Get a list active constraints first.
     * (Constraints that did not fail) */
    const ac = constraints.filter(c => c.isActive());
    for (const constraint of ac) {
        constraint.disable();
    }

    /* (3) Reset per-run information
       (keep values of 'pressed' and 'released'). */
    startPressedTime = undefined;
    pressedTime = undefined;
    movedRightTime = undefined;
    previouslyPressed = t.isKeyDown('right arrow');

    /* (4) Press the green flag and give the program
       some time to Initialize. */
    t.greenFlag();
    await t.runForTime(100);

    /* (5) Get new sprites. */
    sprite = t.getSprite('Sprite1');

    /* (6) Activate information tracking. */
    trackRightKeyCb.enable();
    trackSpriteMoveCb.enable();

    /* (7) Activate constraints. */
    for (const constraint of ac) {
        constraint.enable();
    }

    /* (8) Run the program. */
    await t.runForTime(2000);
}
```

(b) Example code for resetting the program under test (extends Listing 4.2b)

Figure 4.3: Procedure and example code for resetting the program under test

Chapter 5

Implementation

This chapter will provide an overview over Whisker's implementation. Because Whisker is still in early development, some of its implementation may change later.

5.1 Implementation Environment

Whisker uses Scratch 3.0, which is implemented in JavaScript. Therefore, it is also implemented in JavaScript (ES6) for compatibility. One restriction of Whisker's implementation environment is Scratch's dependence on a renderer. In order to operate properly, the Scratch VM needs a HTML canvas to render to. This limits Scratch, and consequently Whisker, to run inside a web environment. Therefore, we use webpack[1] to transpile Whisker's source code into browser-runnable JavaScript.

5.2 General Architecture

One of Whisker's main design goals is to leave the Scratch virtual machine unchanged. There are two reasons for this decision. For one thing, if Whisker used a modified version of Scratch, the modified version would need to be updated regularly as the original version of Scratch receives updates. But the main reason for this decision is that it allows Whisker to be attached to any instance of the Scratch VM. Therefore, anything that runs Scratch 3.0 can theoretically use Whisker.

Figure 5.1 shows Whisker's general architecture. It is designed to be a layer between test code and the Scratch virtual machine. The main class `VMWrapper` makes up a wrapper around the Scratch virtual machine, and its components each implement one part of Whisker's functionality. Test code uses a test driver object, which gives access to the methods that are used for testing. The test driver is used instead of directly interacting with the VM wrapper, because it offers a simpler interface and prevents access to Whisker's private methods.

[1] `https://webpack.js.org/`

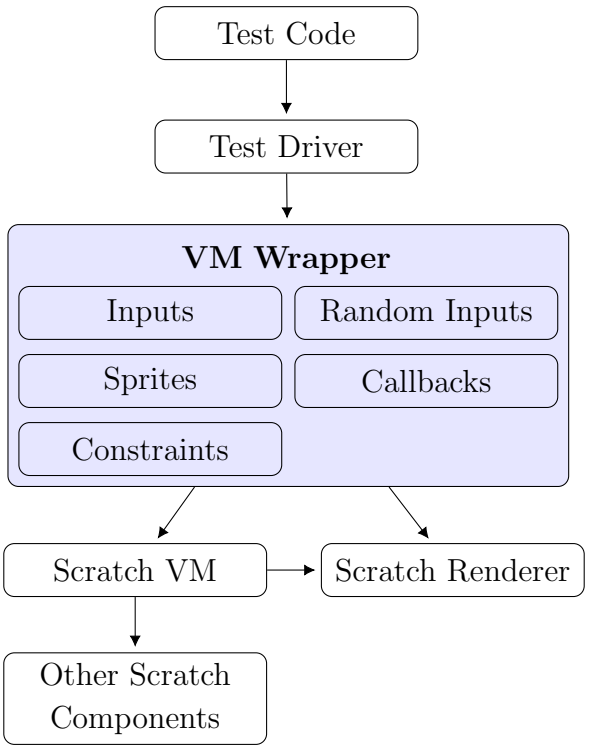


Figure 5.1: General architecture of Whisker

5.3 Scratch Program Execution and the Step Loop

The core of Scratch's virtual machine is a function called `step`, which is called at a constant frequency of 30 times per second using JavaScripts `setInterval` mechanism. It runs the program until a time limit is reached, then it draws the scene using the renderer. If some visual change occurs, i.e. when a sprite changes appearance or position, a redraw is requested, and the program execution step is stopped earlier (in turbo mode redraw requests are ignored). Figure 5.2 shows a simplified version of this procedure.

For the purpose of controlling the execution of the program, and to run multiple functions alongside the step loop, Whisker clears the interval that runs the `step` function and runs its own loop, which in turn calls Scratch's `step` function, instead. Figure 5.3 shows the procedure of Whisker's step function. In addition to running the Scratch program, it performs other tasks which have to be executed periodically.

To make running and pausing the program on demand possible, we implemented a class called `Stepper`, which queues and periodically executes functions in a fixed frequency using JavaScript's intervals as needed. Whenever a test runs the Scratch program, the `Stepper` is used to execute Whisker's own `step` function periodically, until the execution is done. Listing 5.4 shows an excerpt of Whisker's run method, in which the step loop is implemented. Although the Scratch program has to be run asynchronously, we use JavaScript's `Promise` API to make running the program

```
_step () {
    this.threads = this.threads.filter(thread => !thread.isKilled);

    /* Execute the active scripts. */
    this.sequencer.stepThreads();

    /* Render if a renderer is attached. */
    if (this.renderer) {
        this.renderer.draw();
    }

    /* Emit events. */
    ...
}

start () {
    /* Execute the step function 30 times per second. */
    this._steppingInterval = setInterval(() => {
        this._step();
    }, THREAD_STEP_INTERVAL);
}
```

(a) Simplified `step` and `start` functions from Scratch's `Runtime` class

```
stepThreads () {
    WORK_TIME = 0.75 * THREAD_STEP_INTERVAL;
    this.timer.start();

    /* Execute the program until there is no more work,
     * the work time is up, or a redraw is requested. */
    while (numActiveThreads > 0 &&
           timer.timeElapsed() < WORK_TIME &&
           (!this.runtime.redrawRequested || this.runtime.turboMode)) {

        /* Execute each script sequentially. */
        for (thread of this.runtime.threads) {
            /* Execute the thread until either the script's end
               is reached, one loop iteration is executed,
               or the thread yields. */
            this.stepThread(thread);
        }
    }
}
```

(b) Simplified `stepThreads` function from Scratch's `Sequencer` class

Listing 5.2: Simplified Scratch step code

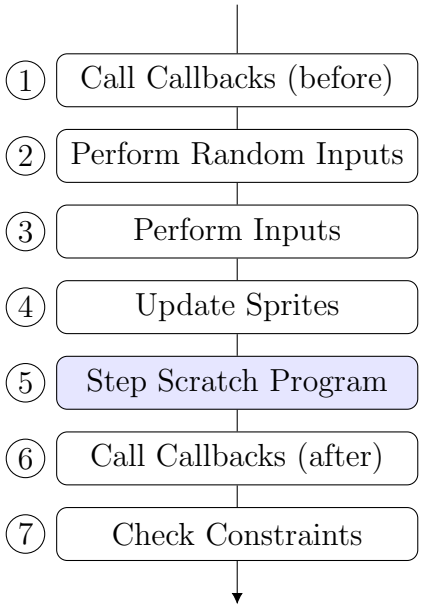


Figure 5.3: Whisker's step procedure

seem like a normal method call. Tests are declared as `async functions`, so they can use the `await` keyword to wait until a run of the program is done. Otherwise, other test code would continue to be executed while the JavaScript engine waits for the next Scratch step. Figure 5.5 presents a code example for this. Whisker prevents multiple runs of the Scratch program at the same time by throwing an error, when a run of the program is being started, while another one is still going on.

```
async run (condition, timeout, steps) {
    ...
    while (this.running &&
           this.getRunTimeElapsed() < timeout &&
           this.getRunStepsExecuted() < steps &&
           !condition()) {
        await this.stepper.step(this.step.bind(this));
        ...
    }
    ...
}
```

Listing 5.4: Excerpt from Whisker's run method

Running other code before and after Scratch's step could theoretically be problematic in a single-threaded environment like JavaScript, because it could take longer than the allowed 1/30 second per step and therefore slow down the Scratch program. But in reality, most Scratch programs will only use a fraction of the allocated work time for the step, because they visually change sprites in every step, which makes the step finish earlier due to the issued redraw request. And even when no redraw

```
const test = async function (t) {
    await t.runForTime(1000);
    console.log('This will be executed after the run ends.');

    t.runForTime(1000);
    console.log('But this will be executed during the run (no await).');
}
```

Listing 5.5: Using JavaScript's Promise API to wait for runs

is requested, the allocated time interval for rendering the picture is long enough to allow Whisker to run its other tasks without exceeding the time limit of 1/30 second. Therefore, Whisker can run other tasks in between the Scratch steps without interfering with the program. In section 6.4, we will prove this empirically by measuring how long Scratch's steps and Whisker's other tasks take to execute.

5.4 Accessing Sprites and Variables

Scratch's implementation differentiates between `Sprites` and `RenderedTargets`. `Sprite` objects contain the scripts, costumes, and sounds of the sprite. They can be seen as a kind of blueprint for the actual sprite instances, which are objects of the `RenderedTarget` class. In contrast, we will call rendered targets sprites, since this resembles the user's view.

Whisker implements a wrapper class around `RenderedTarget` and provides access to these wrappers, instead of the sprite objects themselves. This serves two purposes: Firstly, the wrapper objects offer a simpler and more test-friendly interface. This interface allows to access the sprite's attributes (i.e. the attributes of the `Sprite` object and the `RenderedTarget` object), provides useful helper methods for testing, and gives access to the sprite's variables. Secondly, it saves the sprite's attributes and variable values from the last rendered frame, which is very useful for testing. These attributes are made available in the `old` attribute of the sprite wrapper. The `old` values are updated, i.e. replaced with the current values, before every step of the Scratch program. This is illustrated in Whisker's step procedure shown by Figure 5.3 in step 4.

In order to keep track of the sprite wrappers, Whisker keeps a mapping between the unique id of the original `RenderedTarget` objects and their wrappers. Whenever a sprite is requested, Whisker first checks the mapping for an existing wrapper, and only creates a new one if there isn't already one for the specific sprite. This way, only one wrapper object can exist for every sprite in the program. The same thing is done with variables. Each sprite wrapper keeps track of its own variables.

5.5 Simulating Inputs

Scratch's virtual machine provides an interface to perform input events on the Scratch program through its `postIOData` method. This method is usually used by Scratch's GUI, which forwards input events from the web browser to the virtual machine. But we can use this API for Whisker to simulate mouse and keyboard inputs. However, `postIOData` is not a very user-friendly interface, as it requires extra information about the Scratch stage and coordinate conversion for mouse inputs. Because of this, Whisker takes inputs in a slightly simpler format and converts them to the format that `postIOData` expects. Listing 5.6 shows a mouse input in Whisker's format and the resulting `postIOData` call. Answers to Scratch's `ask` blocks, are simulated in a similar way, but though the VM's event system instead of `postIOData`.

```
t.inputImmediate({
    device: 'mouse',
    isDown: true,
    x: 50,
    y: 100
});
```

(a) A mouse input in Whisker's format

```
vm.postIOData('mouse', {
    isDown: true,
    x: 194.09,
    y: 53.65,
    canvasWidth: 321.25,
    canvasHeight: 241.43
});
```

(b) The resulting `postIOData` call

Listing 5.6: Resulting IO event from a mouse input

Whisker provides methods to either perform inputs immediately, or to queue inputs and execute them after a certain time has passed during the next run of the program. For this purpose, Whisker keeps a list of inputs, which are supposed to be performed later. When the program under test is executed, these inputs are checked before every Scratch step and inputs, which are due, are performed. Figure 5.3 illustrates when the inputs are called in Whisker's step procedure.

Some inputs encompass multiple IO events. For example, pressing a key for a certain duration and then releasing it requires two input events. For this purpose, Whisker keeps a state for each input, which indicates how much of the input was already performed, and what action has to be carried out next.

5.6 Automated Input Generation

We implemented a simple form of automated input generation using of Whisker's randomized inputs. The algorithm detects which inputs the program under test responds to, and registers random inputs accordingly. To determine the inputs,

Whisker simply iterates through the blocks of the currently loaded program and searches for certain block types and block configurations, which detect user inputs. Then it registers a random input for each found configuration.

Table 5.7 lists block configurations, which depend on user inputs, and the resulting inputs, which get registered for each block configuration. Each resulting input depends only on the block type and its parameter. The only exception to this are inputs that answer seeded strings to `ask` blocks. Whisker checks for comparisons between the `answer` variable and string constants. These string constants are then seeded as possible answers for the `ask` blocks.

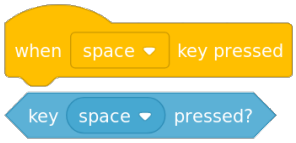
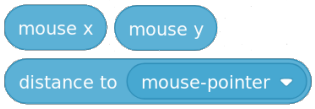
Scratch Block(s)	Resulting Input(s)
	(1) Press the respective keyboard key
	(1) Click the left mouse button
	(1) Move the cursor to a random position
	(1) Move the cursor near / onto the respective sprite (2) Move the cursor to a random position
	(1) Click near / onto the respective sprite (2) Click on a random position
	(1) Click on a random position
	(1) Answer with a randomly generated string
	(1) Answer with the compared string constant

Table 5.7: Scratch's input blocks and their resulting generated inputs

5.7 Callbacks and Constraints

Callbacks and constraints are both implemented as JavaScript functions, that are periodically called during the program execution. When they are registered, they are simply put into a list. Then, during the step procedure, Whisker iterates over both lists and executes every callback and constraint. Both callbacks and constraints can be enabled and disabled, which adds or removes them from their respective list.

Constraints have one key difference to normal callbacks. When constraints are executed, assertion errors are caught. When an error is caught, the constraint is removed from the list, and the error is forwarded to the VM wrapper. This makes it possible to react to failed constraints in different ways. The VM wrapper can, depending on configuration, re-throw the error to fail the test, stop the current run of the program, or ignore the constraint failure entirely.

5.8 Coverage Measurement

Whisker can be used to measure statement coverage on Scratch programs. It does this by temporarily modifying Scratch's `Thread` class. This class has two methods, `pushStack` and `reuseStackForNextBlock`, which put blocks onto the execution stack. Whisker intercepts these methods by replacing them with a version that saves the unique id of the executed blocks and then calls the respective original method. This way, Whisker can track which blocks are executed by the program.

In order to calculate the coverage, we also need a complete list of the program's blocks. To compose this list, Whisker traverses all scripts of the program and saves each block's unique id. Because Whisker traverses the scripts, blocks which are not part of a script with a hat are ignored for the coverage measurement. This is not a problem, since these blocks are not reachable by the program anyways. Whisker considers all reachable control structure blocks, statement blocks and hats for the statement coverage. Coverage is measured per sprite, and program wide coverage can simply be calculated from the per-sprite coverage. Listing 5.8 shows an example coverage report and Listing 5.9 gives a code example of how to measure statement coverage with Whisker.

```
1 { stage: { covered: 0, total: 0 },
2   bowl: { covered: 16, total: 16 },
3   bananas: { covered: 12, total: 24 },
4   apple: { covered: 15, total: 18 } }
```

Listing 5.8: Example coverage report

Currently, coverage measurement can only be performed on Scratch 3.0 (.sb3) projects, since earlier projects don't make the unique id of each block persistent in

the project file. However, this is not a problem since older Scratch projects can simply be converted to Scratch 3.0 projects by opening them in Scratch 3.0 and saving them.

```
/* (1) Replace pushStack and reuseStackForNextBlock,
   so the executed blocks can be tracked. */
CoverageGenerator.prepareThread(Thread);

/* (2) Traverse the scripts of the currently loaded program
   and gather a list of all reachable blocks. */
CoverageGenerator.prepare(vm);

/* Run the Scratch program.
* (Can be run and reloaded multiple times.) */
...

/* (3) Retrieve the current statement coverage. */
const coverage = CoverageGenerator.getCoverage();

/* (4) Restore the original implementations of pushStack and
   reuseStackForNextBlock. */
CoverageGenerator.restoreThread(Thread);
```

Listing 5.9: Example code for coverage measurement using Whisker

In order to measure coverage, one has to gain access to Scratch's `Thread` class. This can be achieved either at compile time or at runtime. Listing 5.10 shows example code of how this can be done.

```
const VirtualMachine = require('scratch-vm');
const Thread = require('scratch-vm/src/engine/thread');
```

(a) At compile time using JavaScript's module system

```
const Thread = vm.runtime.threads[0].__proto__
```

(b) At runtime by getting the class of a running `Thread` instance

Listing 5.10: Acquiring Scratch's Thread class

Chapter 6

Evaluation

In this chapter, Whisker will be evaluated. We will conduct three experiments to answer the following research questions:

RQ1: Can automated testing for Scratch produce test results that match the results of manual grading?

RQ2: How flaky are automated tests for Scratch?

RQ3: What statement coverage can be achieved by controlling Scratch programs with automatically generated input?

RQ4: Does the testing process slow down the program under test?

In the first experiment, we will execute three different test suites on a set of Scratch projects and determine the accuracy of our test results and the flakiness of our test suites. For the second experiment, we will run a variety of projects with automated input generation to find out how much of Scratch programs can be reached by automatically generated input. Finally, in the third experiment we will measure execution times to check if the testing process slows down the program under test.

Section 6.1 describes the programs under test as well as the test suites we will use for the evaluation. It also describes the software and hardware of our testing setup. Then, Sections 6.2, 6.3, 6.4, and 6.5 each describe the results to one research question. Each of the sections describes the experiment we conducted and the indicators we used to answer the respective question. Afterwards, we discuss the results in Section 6.6. Finally, Section 6.7 points out and discusses possible threats to validity.

The evaluation data will be available at `https://github.com/marvinkreis/bachelor-thesis/`.

6.1 Experimental Setup

6.1.1 Testing Environment

Figure 6.1 lists the software and hardware we used to execute test suites on Scratch programs. We used Whisker's web interface, which we described in Section 3.2, for this purpose. All test executions were performed on the same machine. For two parts of the evaluation, specifically to measure execution time and to measure coverage over time, we used slightly modified versions of Whisker as well as its GUI.

Whisker	Whisker 0.2
Scratch VM	scratch-vm 0.2.0-prerelease.20181108204010
Web Browser	Chrome 70.0.3538.110 (64-Bit)
JavaScript	V8 7.0.276.40
OS	Windows 8.1 Version 6.3 (Build 9600)
CPU	Intel Core i5 4670 (4 x 3.40 GHz)
GPU	Nvidia GeForce GTX 1080
RAM	8GB DDR3-1600

Table 6.1: Software and hardware used for evaluation

6.1.2 Projects Under Test

This section will list the Scratch projects we used to evaluate Whisker and explain why we chose these projects.

(P1) Catching Game

The first set of Scratch projects consists of 37 student implementations of a simple catching game. The projects originate from a voluntary Scratch workshop for sixth and seventh grade students [5]. In this workshop, students were introduced to Scratch through several small exercises, building up to the final and most complex exercise to develop the aforementioned catching game. Students were given a Scratch project as a template to fill with their implementation.

In this game, apples and bananas periodically fall from random positions at the top of the screen. The player earns points by catching these fruits with a bowl at the bottom of the screen, which is controlled with the left and right arrow keys. The game ends when 30 seconds have passed or if an apple touches the ground (a game over). A screenshot of the sample implementation of this program can be seen in Figure 6.2 and a summary of the program specification from the assignment is included in Table 6.6. Additionally, the full task description from the workshop (in german) may be found in Figure A.1 in the appendix.

The Scratch projects in this set were manually evaluated and scored using a point-based grading scheme in the range of $[0, 30]$. This enables us to determine the

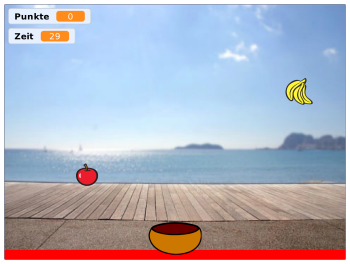


Figure 6.2: Screenshot of the sample implementation

quality of our test results by using the manually assigned scores as ground truth, and comparing the results of automated testing to them. Table 6.12 contains the manual scores for each project and Table A.2 shows the grading scheme used to determine the scores.

We use these projects to evaluate the accuracy of our testing approaches. The projects are very suitable for this task because of two reasons. Firstly, there exists a relatively precise specification for the programs, which enables us to test the implementations according to this specification. And secondly, the student solutions are of varying degrees of quality, which is desirable for the evaluation, because we can analyze a wide range of different outcomes.

However, since the Scratch programs were not written for the purpose of automated testing, some of the projects were unusable for our evaluation. Therefore, we excluded 6 of the 37 projects in our statistics. Table 6.3 shows the excluded projects together with the reasons for excluding them. Most of the projects were excluded because they don't start properly. Some of the projects start with a key press instead of the green flag, others go game over when they are started the first time, but work in subsequent runs. Since the specification did not actually specify that the programs have to start with the green flag, the manual scoring ignores these problems, but our tests don't. Hence, these projects would create a discrepancy between manual scores and test results, which only exists due to the difference in scoring systems. We believe the statistics convey our results more clearly without these projects.

(P2) Code Club Projects

The second set of Scratch projects consists of sample solutions to 24 of Code Club's [8] Scratch exercises. Code Club offers free coding projects and step by step guides for programming beginners. For each project, a sample solution is provided by a volunteer.

We use these projects to evaluate Whisker's automated input generation. The projects work well for this task since they implement a variety of different programs with different input methods. Table 6.4 displays the input methods the projects use.

Project	Reason
Detected through test reports	
K7_S07	Resets all sprite positions when the "a" key is pressed. Automated test input generation picks up on this and simulates presses on the "a" key.
Detected through zero statement coverage	
K6_S01	Starts on up arrow key press instead of green flag.
K6_S06	Starts on space key press instead of green flag.
K6_S14	Starts on space key press instead of green flag.
Detected manually	
K6_S20	Goes game over when green flag is pressed the first time.
K7_S18	Goes game over on start unless sprites are re-positioned manually in the editor.

Table 6.3: Excluded projects from P1 and the reasons for their exclusion

We chose the sample solutions of the exercises since measuring coverage on working implementation will yield better results. Broken implementations are more likely to have unreachable code, because some code might depend on a feature that is not implemented correctly.

Project	# Sprites	# Scripts	# Blocks	Mouse	Keyboard	Ask-Answer
Archery	2	3	21	×	✓	×
Balloons	2	4	26	✓	×	×
Beat The Goalie	3	6	30	×	✓	×
Boat Race	3	3	27	✓	×	×
Brain Game	4	19	76	✓	×	✓
Catch The Dots	5	11	82	×	✓	×
Chat Bot	2	2	26	✓	×	✓
Clone Wars	7	17	76	×	✓	×
Create Your Own World	13	26	165	×	✓	×
Dodgeball	5	10	78	×	✓	×
Ghostbusters	5	11	58	✓	×	×
Green Your City	7	8	52	×	✓	×
Lost In Space	6	4	24	×	×	×
Memory	6	11	58	✓	×	✓
Moonhack Scratch 2017	3	4	27	×	✓	×
Poetry Generator	4	2	18	✓	×	✓
Rock Band	5	6	18	✓	×	×
Snowball Fight	4	7	37	×	✓	×
Space Junk	8	13	68	×	✓	×
Sprint	5	9	78	×	✓	×
Synchronised Swimming	2	7	23	×	✓	×
Tech Toys	9	5	25	✓	✓	×
Username Generator	3	2	5	✓	×	×
Paint Box	9	14	42	✓	×	×

Table 6.4: Block counts and input methods of the Code Club projects (P2)

6.1.3 Test Suites

The set of catching game projects (P1) will be tested with three different test suites in order to analyze the quality of their test results. Table 6.6 lists which part of the specification is covered by each test suite. Additionally, the number of tests or

constraints, and the size of the test suite measured in source lines of code (SLOC), can be found in Table 6.5. In this section, we will describe each of the test suites.

Test Suite	Type	#Tests / #Constraints	SLOC
T1	Normal	28 Tests	852
T2	Constraint	26 Constraints	620
T3	Constraint	26 Constraints	618

Table 6.5: Number of tests and source lines of code per test suite

(T1) Normal Test Suite

This test suite follows an ordinary testing approach. Each test case runs the program once in order to check one part of the specification, mostly without the use of constraints. Tests simulate inputs deliberately and mostly perform assertions between runs of the program. The test suite encompasses 28 test cases, which correspond to the specification in Table 6.6.

For many test cases, the catching game has to be played without letting an apple touch the ground for a period of time. In order to accomplish this, we implemented a function that takes the x coordinates of the apple and bowl sprites, and simulates arrow key presses to move the bowl towards the apple's position. This function, as well as a usage example, are shown in Listing 6.7.

(T2) Constraint Test Suite

This test suite implements the constraint testing approach from Chapter 4. With this test suite, we tried to test the program with only one execution in total. Therefore, this test suite only has a single test case, which registers 26 constraints to check various parts of the specification (corresponding to Table 6.6).

This test case deliberately simulates inputs with the goal of winning the game. We use the same helper method as test suite T1 to simulate key presses depending on the bowl's and the apple's position. Because of this, the bowl should always catch the apple in working implementations, and the program should never go game over because of an apple touching the ground. Therefore, we don't check the interaction between the apple sprite and the ground with this test suite.

(T3) Random Input Test Suite

This test suite uses 26 constraint, which are mostly the same T2's. It uses Whisker's automated input generation to control the program with random inputs, and resets the program multiple times in order to increase the likelihood of multiple apples getting caught in one of the runs. But since it is nevertheless very unlikely to have

#	Specification	Covered?, Test Number					
		T1		T2		T3	
	Initialization						
1	Timer starts at 30 seconds and score starts at 0	✓	1	✓	1	✓	1
2	Bowl starts at $X = 0$ / $Y = -145$	✓	2	✓	2	✓	2
3	Fruits have a size of 50%	✓	3	✓	3	✓	3
	Bowl Movement						
4	Bowl moves left/right when the corresponding arrow key is pressed	✓	4	✓	4	✓	4
5	Bowl can only move horizontally with a speed of 10	✓	5	✓	5	✓	5
	Fruit Falling						
6	Apples fall down	✓	6	✓	6	✓	6
7	Apples fall in a straight line with a speed of -5	✓	7	✓	7	✓	7
8	Bananas fall down	✓	8	✓	8	✓	8
9	Bananas fall in a straight line with a speed of -7	✓	9	✓	9	✓	9
	Fruit Spawn						
10	Apples spawn again at the top of the screen after touching the bowl	✓	10	✓	10	✓	10
11	Apples spawn at random X positions	✓	11	✓	11	✓	11
12	Apples spawn at $Y = 170$	✓	12	✓	12	✓	12
13	Bananas spawn again at the top after touching the bowl or the ground	✓	13	✓	13	✓	13
14	Bananas spawn at random X positions	✓	14	✓	14	✓	14
15	Bananas spawn at $Y = 170$	✓	15	✓	15	✓	15
16	Only one apple must fall down at a time	✓	16	✓	16	✓	16
17	Only one banana must fall down at a time	✓	17	✓	17	✓	17
18	Banana must wait for a second before falling down in the beginning	✓	18	✓	18	✓	18
19	Banana must wait for a second before falling down after displaying ”-8”	✓	19	✓	19	✓	19
	Fruit Interaction						
20	Apple gives 5 points when it touches the bowl	✓	20	✓	20	✓	20
21	Game over when the apple touches the ground	✓	21	×		✓	21
22	Apple displays ”Game Over!” message when it touches the ground	✓	22	×		✓	22
23	Banana gives 8 points when it touches the bowl	✓	23	✓	21	✓	23
24	Banana subtracts 8 points when it touches the ground	✓	24	✓	22	✓	24
25	Banana displays ”-8” message when it touches the ground	✓	25	✓	23	✓	25
	Timer						
26	Timer is decremented by one once a second	✓	26	✓	24	✓	26
27	Game stops after 30 seconds elapsed	✓	27	✓	25	×	
28	Bowl must display ”Ende!” message after 30 seconds elapsed	✓	28	✓	26	×	

Table 6.6: Project specification and each test suite’s coverage of the specification

```
const followSprite = function (t, bowlX, spriteX) {
    /* Stop if the bowl is near enough. */
    if (Math.abs(bowlX - spriteX) <= 10) {
        t.inputImmediate({device: 'keyboard', key: 'left arrow',  isDown: false});
        t.inputImmediate({device: 'keyboard', key: 'right arrow', isDown: false});

    } else if (bowlX > spriteX) {
        t.inputImmediate({device: 'keyboard', key: 'right arrow', isDown: false});
        t.inputImmediate({device: 'keyboard', key: 'left arrow',  isDown: true});

        /* Trick "when key pressed" hats to fire by letting go of the key and immediately pressing it again. */
        t.inputImmediate({device: 'keyboard', key: 'left arrow',  isDown: false});
        t.inputImmediate({device: 'keyboard', key: 'left arrow',  isDown: true});

    } else if (bowlX < spriteX) {
        t.inputImmediate({device: 'keyboard', key: 'left arrow',  isDown: false});
        t.inputImmediate({device: 'keyboard', key: 'right arrow', isDown: true});

        /* Trick "when key pressed" hats to fire by letting go of the key and immediately pressing it again. */
        t.inputImmediate({device: 'keyboard', key: 'right arrow', isDown: false});
        t.inputImmediate({device: 'keyboard', key: 'right arrow', isDown: true});
    }
};

const testSomething = function (t) {
    ...
    /* Catch apples with the bowl during runs. */
    t.addCallback(() => {
        apple = getNewestClone(apple);
        followSprite(t, bowl.x, apple.x);
    });
    ...
};
```

Listing 6.7: Code for simulating arrow key presses for the catching game

the apple not touch the ground at all with random inputs, we don't for check for a game over after 30 seconds, and only let the program run for 10 seconds a time. We chose to reset the program 30 times during the test, resulting in an execution time of $30 \times 10s = 300s$. Listing 6.8 shows the exact configuration we used to simulate input for the test case. We chose to simulate random inputs with a duration between 50ms and 100ms each 150ms.

```
t.setRandomInputInterval(150);
t.detectRandomInputs({duration: [50, 100]});
```

Listing 6.8: Code for automated input generation in test suite T3 (random input)

6.2 RQ1: Accuracy of Test Results

In this section, we will answer the following research question:

> **RQ1: Can automated testing for Scratch produce test results that match the results of manual grading?**

In order to answer this question, we executed test suites T1-T3 on the catching game projects (P1) ten times, and analyzed the results. To be useful for grading, test results have to indicate the same grades, which one would assign the programs

through manual assessment. Intuitively, a higher score means that more tests (or constraints) should pass, while a lower score should result in fewer test (or constraint) passes. Therefore, we use the correlation between the number of test passes and the manual scores as an indicator. We measure the correlation through Pearson's correlation coefficient, which is denoted as r. r can take on values in the range of $[-1, 1]$. A high correlation coefficient indicates a strong relationship between two values, while a low correlation coefficient indicates a weak relationship. Usually, a value of $r \geq 0.5$ is considered to indicate a strong correlation. Negative values indicate a reverse relationship, which is not relevant for our observations.

Null hypothesis (H_0): The number of test passes and the results of manual scoring only show a weak correlation or no correlation at all.

Alternative hypothesis (H_1): The number of test passes closely match manually assigned scores. They have a strong correlation.

Normal Test Suite (T1)

Figure 6.9 compares each project's number of test passes of test suite T1 to its manual score. Possible values for the manual scores are in the range of $[0, 30]$ and possible values for the number of test passes are in the range of $[0, 28]$. The blue line displays the linear regression. We can observe a strong correlation between the number of passing test cases and the manual scores, with a correlation coefficient of $r = 0.893$ (p-value = 1.45e-11) for the first run and $r = 0.899$ (p-value = 6.59e-12) for the average number of passes over ten runs. Except for few irregularities, the test results closely resemble the manually assigned scores.

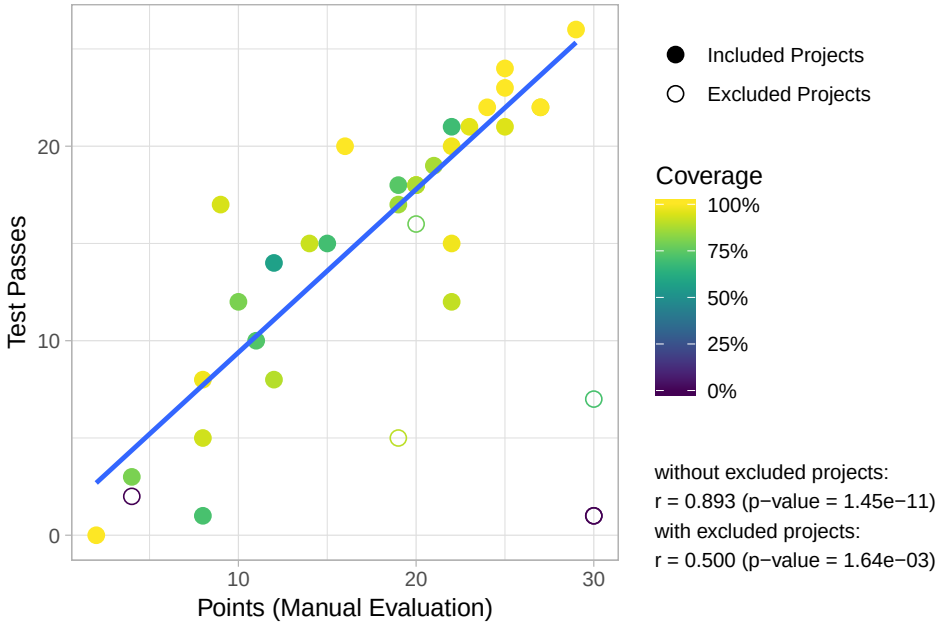
Constraint Test Suite (T2)

Figure 6.10 compares each project's number of passing constraints of test suite T2 to its manual score. Possible values for the number of constraint passes are in the range of $[0, 26]$. We can again observe a strong correlation between the two scores, with a correlation coefficient of $r = 0.878$ (p-value = 9.06e-11) for the first run and $r = 0.889$ (p-value = 2.43e-11) for the average number of passes over ten runs.

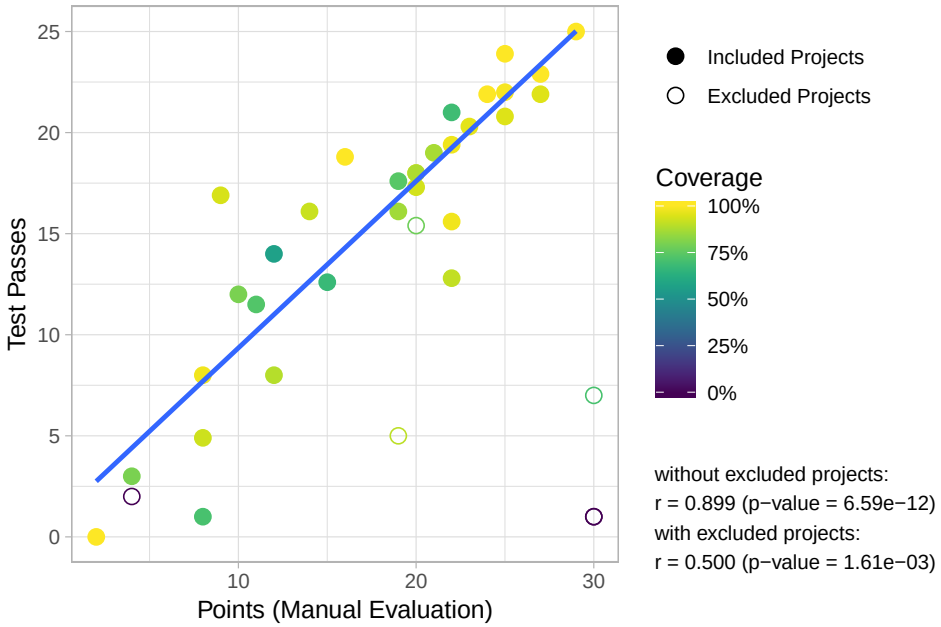
Random Input Test Suite (T3)

Figure 6.10 compares the constraint passes of test suite T3 to the manual scores. Possible values for the number of constraint passes are in the range of $[0, 26]$. The results indicate a strong correlation with a correlation coefficient of $r = 0.869$ (p-value = 2.39e-10) for the first run and $r = 0.882$ (p-value = 5.23e-11) for the average over ten runs.

Each project's manual scores and number of test passes can be found in Table 6.12.

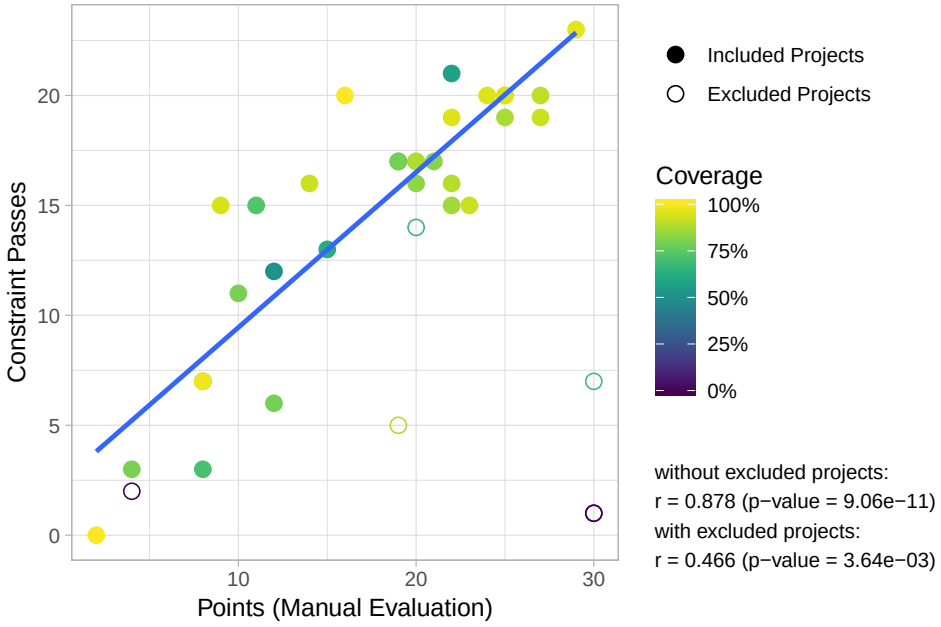


(a) First run

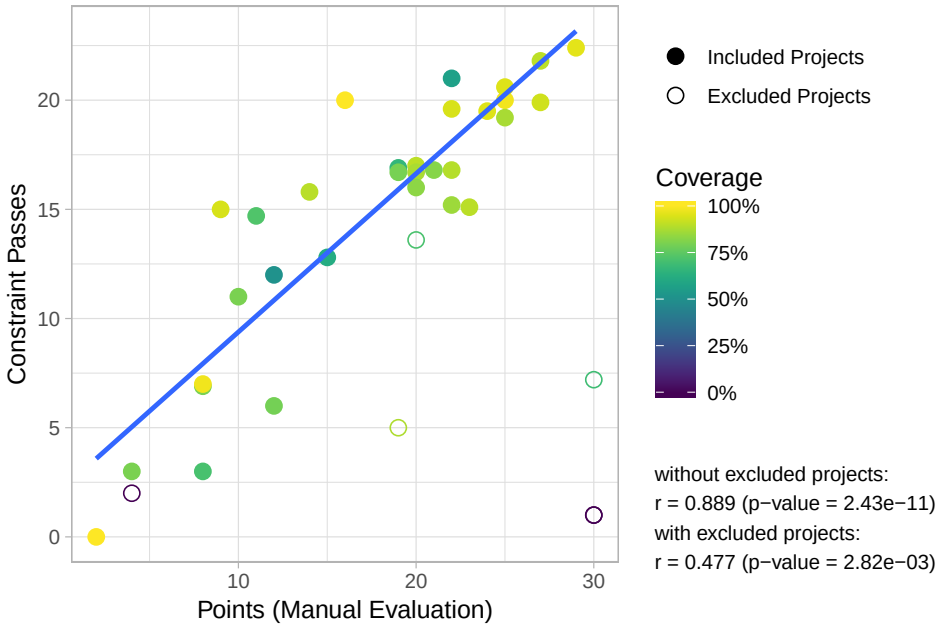


(b) Average over ten runs

Figure 6.9: Comparison between test results of test suite T1 (normal) and manually assigned scores

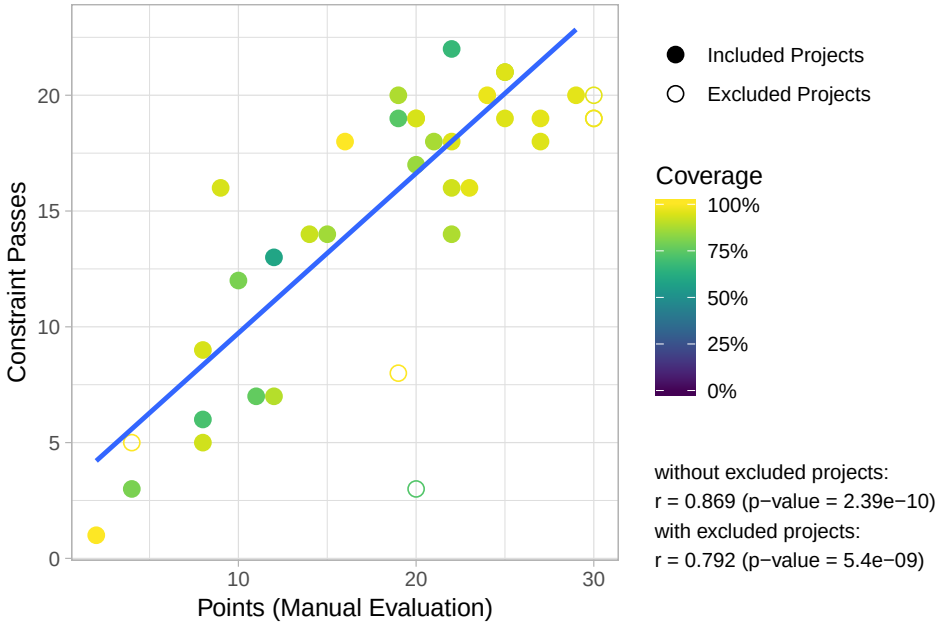


(a) First run

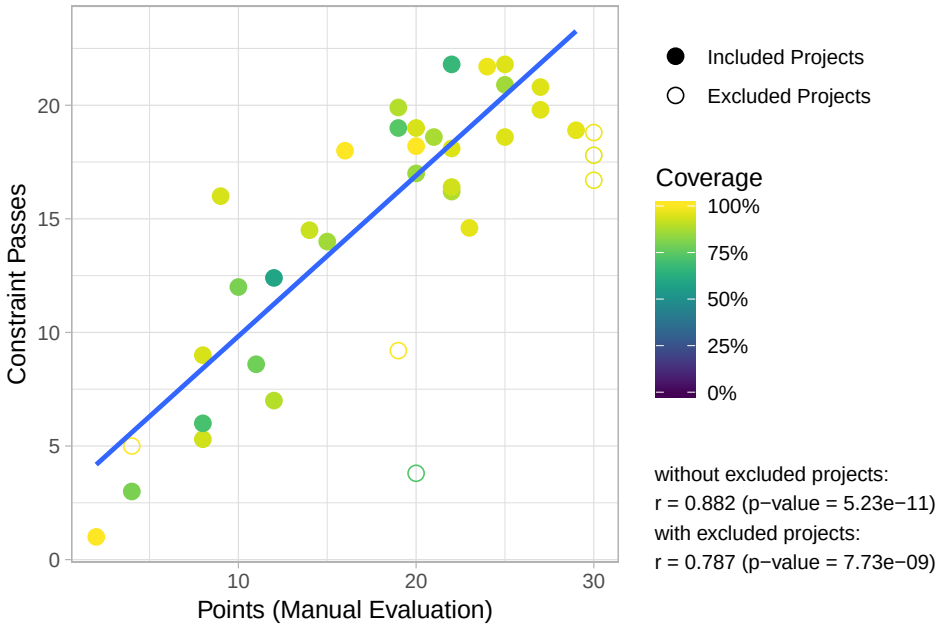


(b) Average over ten runs

Figure 6.10: Comparison between test results of test suite T2 (constraint) and manually assigned scores



(a) First run



(b) Average over ten runs

Figure 6.11: Comparison between test results of test suite T3 (random input) and manually assigned scores

Project	Excl.	Points	T1 1st	T1 avg	T2 1st	T2 avg	T3 1st	T3 avg
K6_S01	✓	30	1	1.0	1	1.0	20	17.8
K6_S02	×	15	15	12.6	13	12.8	14	14.0
K6_S03	×	11	10	11.5	15	14.7	7	8.6
K6_S05	×	23	21	20.3	15	15.1	16	14.6
K6_S06	✓	30	1	1.0	1	1.0	19	16.7
K6_S10	×	22	20	19.4	19	19.6	18	18.1
K6_S11	×	8	5	4.9	7	6.9	5	5.3
K6_S13	×	8	8	8.0	7	7.0	9	9.0
K6_S14	✓	4	2	2.0	2	2.0	5	5.0
K6_S15	×	10	12	12.0	11	11.0	12	12.0
K6_S16	×	2	0	0.0	0	0.0	1	1.0
K6_S18	×	27	22	21.9	19	19.9	19	19.8
K6_S19	×	25	23	22.0	20	20.0	21	20.9
K6_S20	✓	30	7	7.0	7	7.2	19	18.8
K6_S27	×	4	3	3.0	3	3.0	3	3.0
K6_S29	×	12	8	8.0	6	6.0	7	7.0
K6_S30	×	20	18	18.0	17	17.0	19	18.2
K6_S31	×	16	20	18.8	20	20.0	18	18.0
K6_S33	×	20	18	17.3	17	16.7	19	19.0
K7_S02	×	19	18	17.6	17	16.9	19	19.0
K7_S03	×	14	15	16.1	16	15.8	14	14.5
K7_S04	×	12	14	14.0	12	12.0	13	12.4
K7_S05	×	22	21	21.0	21	21.0	22	21.8
K7_S06	×	20	18	18.0	16	16.0	17	17.0
K7_S07	✓	20	16	15.4	14	13.6	3	3.8
K7_S08	×	21	19	19.0	17	16.8	18	18.6
K7_S10	×	25	21	20.8	19	19.2	21	21.8
K7_S11	×	8	1	1.0	3	3.0	6	6.0
K7_S12	×	27	22	22.9	20	21.8	18	20.8
K7_S14	×	24	22	21.9	20	19.5	20	21.7
K7_S15	×	25	24	23.9	20	20.6	19	18.6
K7_S16	×	19	17	16.1	17	16.7	20	19.9
K7_S17	×	29	26	25.0	23	22.4	20	18.9
K7_S18	✓	19	5	5.0	5	5.0	8	9.2
K7_S19	×	22	12	12.8	15	15.2	14	16.2
K7_S20	×	9	17	16.9	15	15.0	16	16.0
K7_S26	×	22	15	15.6	16	16.8	16	16.4

Table 6.12: Manual scores and number of test passes of each project in P1

6.3 RQ2: Flakiness of Tests

In this section, we will answer the following research question:

RQ2: How flaky are automated tests for Scratch?

In automated testing, test cases can yield different results when executed multiple times. Test cases like this are known as *flaky tests*. This inconsistency of test results can either be caused by the program under test or by the test case itself. We determined the number of test-project combinations with inconsistent results over the ten test executions per test suite we performed for RQ1. We use their percentage of all test-project combinations to indicate the flakiness of our test suites.

In order to analyze which test cases where inconsistent for each project, and vice versa, we constructed matrices over the test cases (or constraints) in T1-T3 and the projects in P1. If a test case (or constraint) showed inconsistent results on a certain project over the ten test executions, we assigned a 1 to the respective cell in the matrix, otherwise we assigned a 0. The matrix's sum of rows then shows how may projects had inconsistent test results for each test case, and the sum of columns shows the number of test cases with inconsistent outcomes for each project. Table 6.13 shows an excerpt from one of the resulting matrices for demonstration. The full matrices can be found in Tables A.3, A.4, and A.5 in the appendix. We chose to ignore skipped tests and only assigned a 1 if the test case had a passing and a failing outcome on the project. Tests were skipped if the feature, which is tested by the test, could not be checked properly, for example if too few apples spawned to determine if their spawn positions are random, or if no banana touched the bowl while checking if the banana touching the bowl gives the player points.

	12	13	14	15	16	17
K6_S02	0	**1**	0	0	0	0
K6_S03	0	**1**	0	0	0	0
K6_S05	0	0	0	0	0	0
K6_S10	0	**1**	**1**	0	0	0
K6_S11	0	0	0	0	0	0
K6_S13	0	0	0	0	0	0

Figure 6.13: Excerpt from the test-project matrix of test suite T1's inconsistencies

Normal Test Suite (T1)

Figure 6.14a displays the number of inconsistent projects per test case of T1, and Figure 6.14b displays the number of inconsistent test cases per project. The horizontal lines show the average number of inconsistencies. We can observe that some test cases and some projects tend to be rather inconsistent, with a maximum of 6 inconsistent projects (19.35% of projects) per test case and 5 inconsistent test cases (17.86% of test cases) per project. However, on average only 1.28 projects per

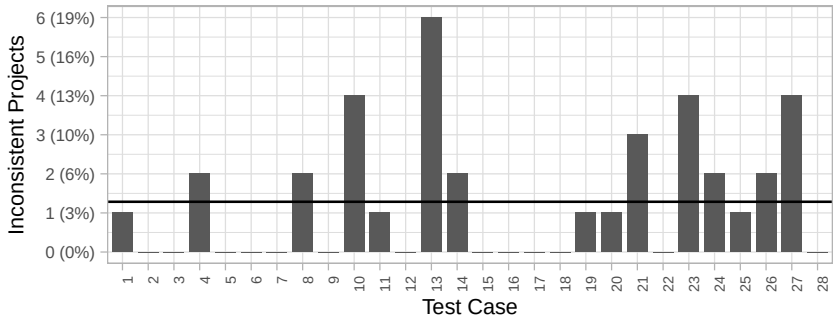
test case and 1.65 test cases per project show inconsistent results. This amounts to 4.15% of all test-project combinations.

Constraint Test Suite (T2)

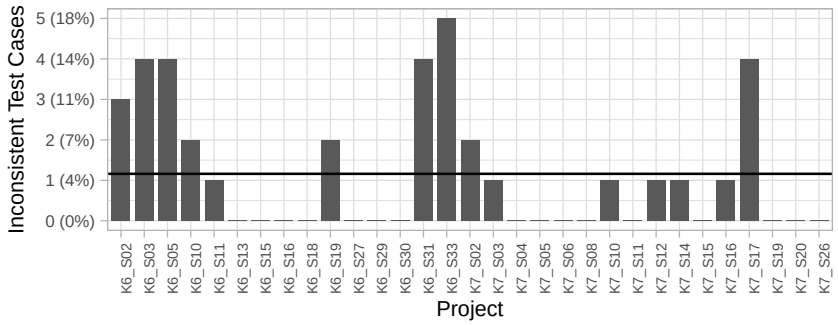
Figure 6.15 shows the number of inconsistencies for test suite T2. We observe fairly similar results to T1. The data shows a maximum of 7 inconsistent projects (22.58% of projects) per constraint and 4 inconsistent constraints (15.38% of constraints) per project. On average, 0.96 projects per constraint and 0.86 constraints per project show inconsistent results, which amounts to 3.10% of all constraint-project pairs.

Random Input Test Suite (T3)

Figure 6.16 shows the number of inconsistencies for test suite T3. We observed a maximum of 9 inconsistent projects (29.03% of projects) per constraint and 6 inconsistent constraints (23.08% of constraints) per project. On average, 1.96 projects per constraint and 1.65 constraints per project show inconsistent results, which amounts to 6.32% of all constraint-project pairs. Since this test suite is non-deterministic, this increase in inconsistent outcomes is expected.

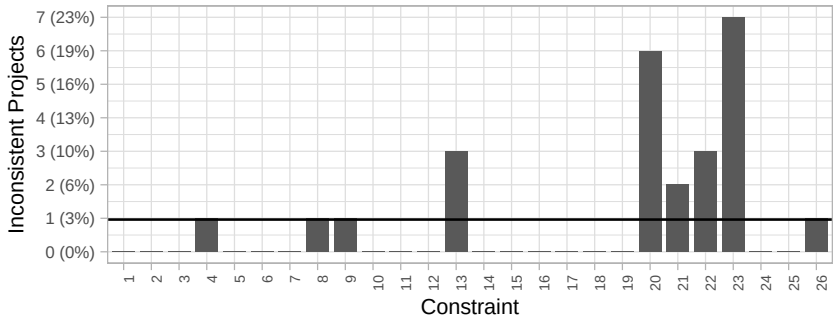


(a) Number of projects with inconsistent outcomes, per test case

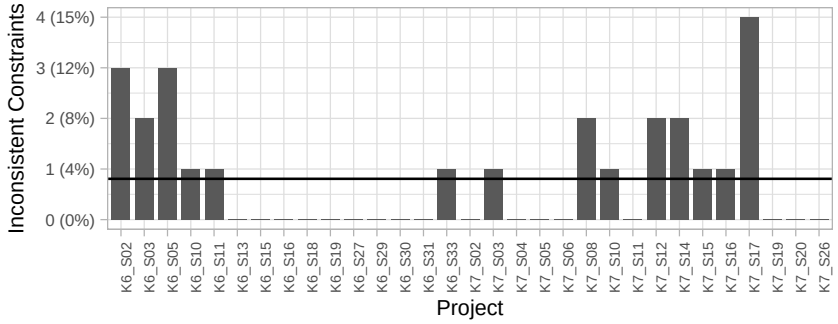


(b) Number of test cases with inconsistent outcomes, per project

Figure 6.14: Inconsistent outcomes of test suite T1 over ten runs

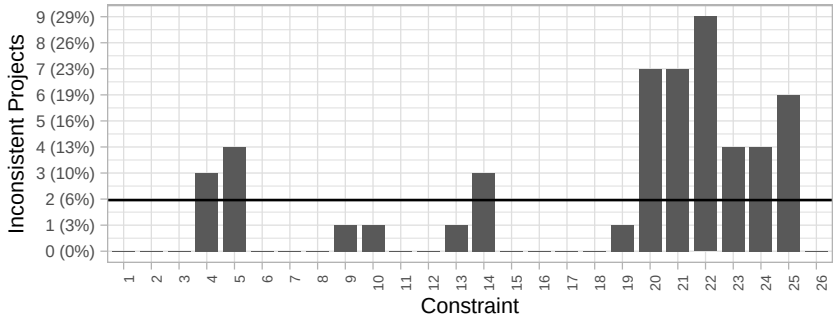


(a) Number of projects with inconsistent outcomes, per constraint

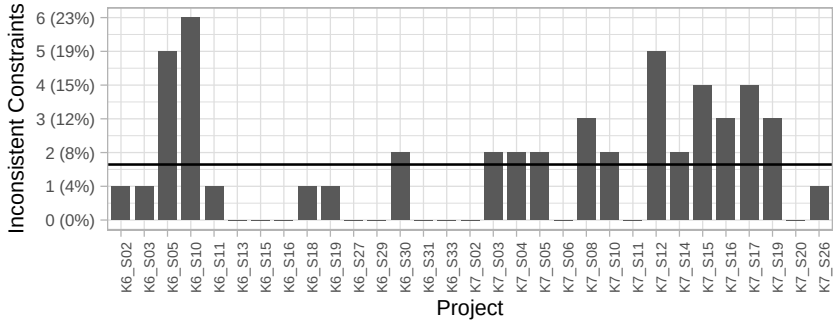


(b) Number of constraints with inconsistent outcomes, per project

Figure 6.15: Inconsistent outcomes of test suite T2 over ten runs



(a) Number of projects with inconsistent outcomes, per constraint



(b) Number of constraints with inconsistent outcomes, per project

Figure 6.16: Inconsistent outcomes of test suite T3 over ten runs

6.3.1 Causes for Inconsistent Test Outcomes

Inconsistent test outcomes can either be caused by the test case (or constraint) or by the project under test. Therefore, we analyzed some of the inconsistencies of test suite T1 and their causes in Table 6.17. Most of them are caused by projects behaving in a non-deterministic manner, tough some are caused by test cases themselves. We noticed that tests, which deal with the banana sprite, i.e. test cases $13, 14, 23, 24, 25$, are relatively inconsistent. We observed that the banana sprite is often poorly implemented, which causes it to behave inconsistently. We suspect that students did not pay much attention to the banana sprite, since it is not essential to play the game. A banana only subtracts points when it touches the ground, while an apple ends the game in this situation.

Project	Test	Cause	Description
K6_S11	1	Both	Test executes the program until each variable has been initialized to the correct value or until 500ms passed, and then checks the variable values. The project incorrectly increments the score variable continually when the program runs. Most of the time, the score variable has the value 0 when the rest of the program has been initialized, but sometimes the variable already gets incremented during initialization.
K7_S16	4	Project	Program sometimes goes game over directly after being started.
K6_S03	8	Project	Program moves banana to a random position on the ground when the right arrow key is pressed. Therefore the test has different outcomes depending on if the apple spawns left or right of the bowl.
K6_S33	8	Both	Test waits for 250ms to let the program initialize, then waits for a banana to appear at the top of the screen. K6_S33 lets the banana fall instantly instead of waiting for a second (as specified), causing the banana to be below the tests threshold for detecting the banana ($y > 100$) after 250ms. The banana then proceeds to re-appear at the top of the screen, but sometimes the banana stays there because an apple already touched the ground, ending the game.
K6_S10	13	Project	Program checks if the apple touches the ground with Scratch's "touching edge" primitive. Therefore, the program sometimes goes game over when an apple randomly spawns too much to the side of the screen and touches a vertical edge.
K7_S03	13	Project	Program does not move banana to the top of the screen when it touches the ground, only when it touches the bowl. Therefore, the banana stays on the ground and is moved only when bowl moves over it, making banana re-spawns dependent on apple spawn locations.
K6_S10	14	Test	Test checks if bananas spawn at random x coordinates. The program uses clones for fruit sprites. The test detects the original sprite and the clone as two banana instances at the same x coordinate and concludes that bananas don't spawn at random x coordinates.
K7_S12	14	Test	Test failed because it detected two fruits spawning at the same x coordinate. In order to speed up test executions, we decided to wait for only two bananas to spawn and compare their positions. The test either may have detected the same fruit twice, or two fruit spawned at the same position. Both will result in the first two bananas being detected at the same x coordinate, failing the test.
K7_S17	19	Project	Programs are supposed to check if the apple touches the color of the red stripe at the bottom to detect when the apple touches the ground. K7_S17 checks for the wrong shade of red which is only present in some places, at the transition of the background and the red line. Therefore, sprites are sometimes not detected touching the ground, and simply stay at the bottom of the screen.
K6_S05	20	Project	To move the banana to a random position whenever it spawns, the program uses Scratch's builtin "go to random position", which moves it to a random y coordinate as well. Afterwards, it immediately adjusts the bananas y coordinate. But for a split second, the banana may touch the bowl or the ground without interaction, which the test detects.
K7_S02	21	Both	Program moves the apple to a random position on the ground 2 seconds after game over, causing the test to recognize it as not game over. The test could wait longer before performing checks to circumvent this problem.
K7_S10	21	Project	Banana sometimes re-appears one more time after game over.
K6_S19	23	Test	The test sometimes detects the banana touching the both the ground and the bowl as the it just touching the bowl.
K6_S02	27	Both	Banana can get stuck in the ground and continually subtract points for a short time after the time is up.
K6_S19	27	Both	Banana sometimes still falls for a short time after the time is up.
K7_S02	27	Test	Test checks if the program went game over by observing for one second if the variables and the sprites' x coordinates change. Therefore, if fruit is currently falling while the game over is checked, the test may see a running game as game over, since only the sprites' y coordinates change.

Project	Test	Cause	Description	Project	Test	Cause	Description
K7_S14	11	Test	See K7_S12, test 14.	K7_S17	23	Test	See K6_S19, test 23.
K6_S02	13	Project	See K7_S03, test 13.	K6_S05	24	Project	See K6_S05, test 20.
K6_S03	13	Project	See K6_S03, test 8.	K7_S17	24	Project	See K7_S17, test 19.
K7_S17	21	Project	See K7_S17, test 19.	K6_S05	25	Project	See K6_S05, test 20.
K6_S05	23	Project	See K6_S05, test 20.	K7_S03	27	Test	See K7_S02, test 27.

Table 6.17: Explanations for inconsistent outcomes of test suite T1

6.4 RQ3: Automated Input Generation

In this section, we will answer the following research question:

> **RQ3: What statement coverage can be achieved by controlling Scratch programs with automatically generated input?**

For RQ1, we already showed that achieving accurate test results with random input is possible. In order to get good test results generally, the generated input needs to be able to reach as much of the program under test as possible. Intuitively, the more of a program gets executed, the more of the its functionality can be checked by tests running on it. Therefore, we want to find out how much of typical Scratch programs' functionality can be reached by controlling it through Whisker's generated input. In order to measure how much of the program is reached, we will use the achieved statement coverage as an indicator.

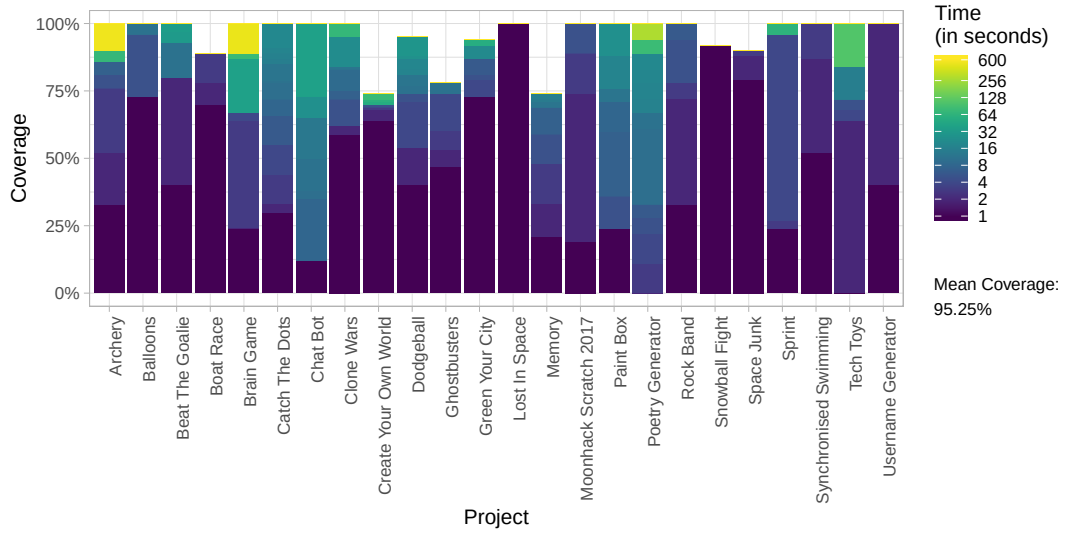
We ran Whisker's automated input generation algorithm on each of the Code Club Scratch projects (P2) for ten minutes, and measured the current statement coverage every second during the execution. We chose not to reset the programs during the execution time, since some of Code Club's projects require much execution time to be covered. In fact, we also tried a configuration of ten minutes with four resets (4×2.5min) and got slightly worse results. We use the mean final statement coverage on the projects to indicate how much of the programs is reached by the automated input. To get more consistent results, we ran this experiment ten times, and took the average of the final coverage scores. Listing 6.18 shows the code, which we used to run the programs. We chose to simulate random inputs with a duration between 100ms and 1000ms each 250ms. In order to access the coverage during the execution we implemented a function called `logCoverage`, which saves the current statement coverage of the program and outputs all saved coverage records once the test is over.

```
1 const measureCoverage = async function (t) {
2     t.detectRandomInputs({duration: [100, 1000]});
3     t.setRandomInputInterval(250);
4
5     for (let timestamp = 1000; timestamp <= 600000; timestamp += 1000) {
6         await t.runUntil(() => t.getTotalTimeElapsed() >= timestamp);
7         t.logCoverage(timestamp, t.isProjectRunning());
8     }
9 }
```

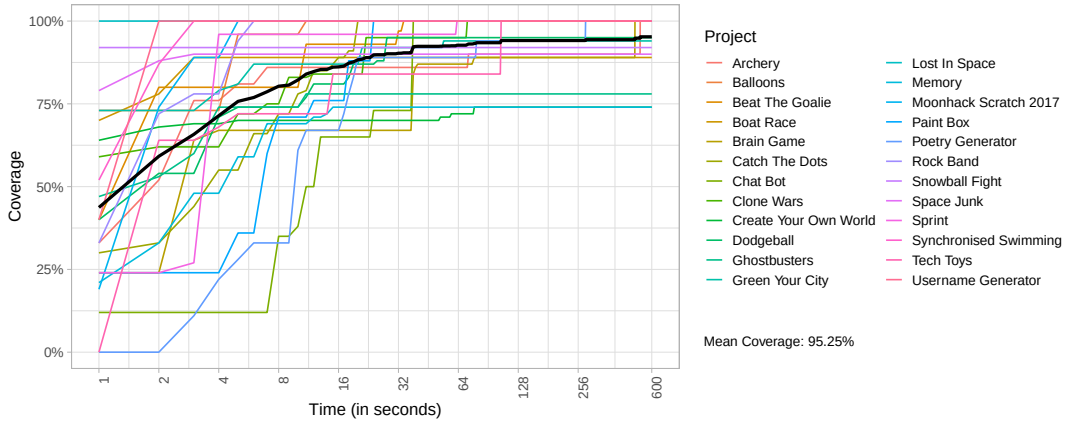
Listing 6.18: Code to measure the coverage of automatically generated input

Figure 6.19 shows the statement coverage we were able to achieve on the Code Club projects (P2) with Whisker's automatically generated input on the first run,

as well as the execution time at which the coverage was achieved. The black line in Figure 6.19b displays the mean coverage of the projects. After ten minutes of run time, we were able to achieve a mean coverage of 95.25% with the lowest coverage for a project being 74% and the highest being 100%. We repeated this experiment ten times and got similar results, with the mean final coverage over ten runs being 95.25%. The mean coverage from ten runs can be seen in Figure 6.20. In comparison, Figure 6.21 and Figure 6.22 show the achieved coverage without simulating any inputs. Here, we only got an average of 47.04% per project on our first run, and 47.14% averaged over ten runs.

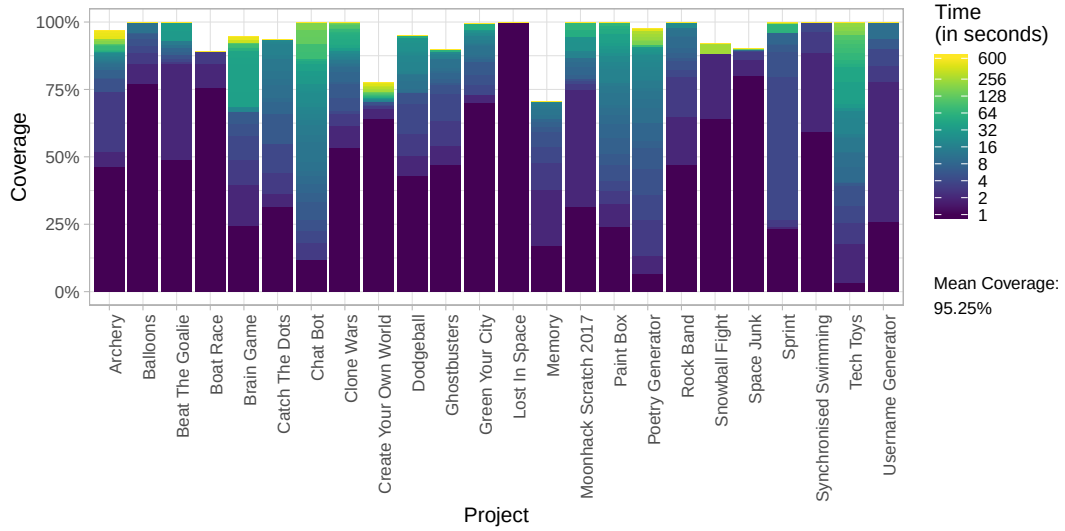


(a) Coverage per project

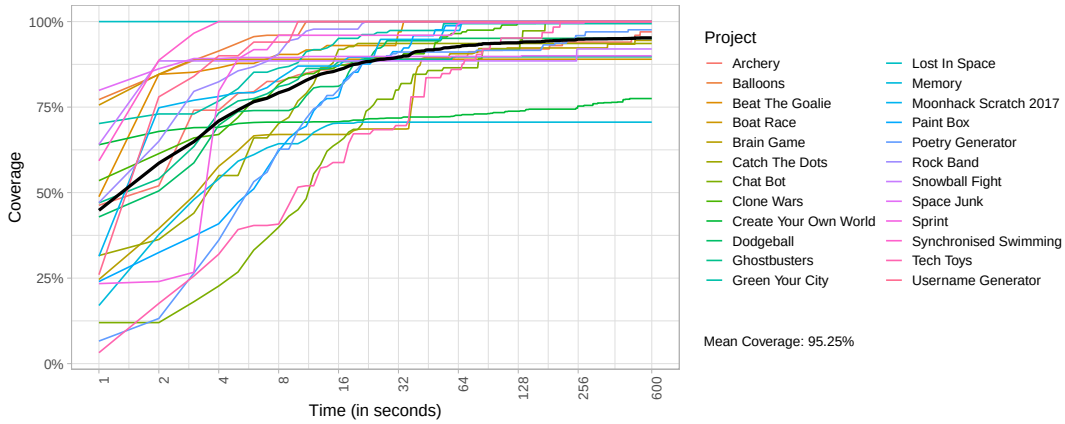


(b) Coverage over time

Figure 6.19: Achieved coverage with automated input on the first run

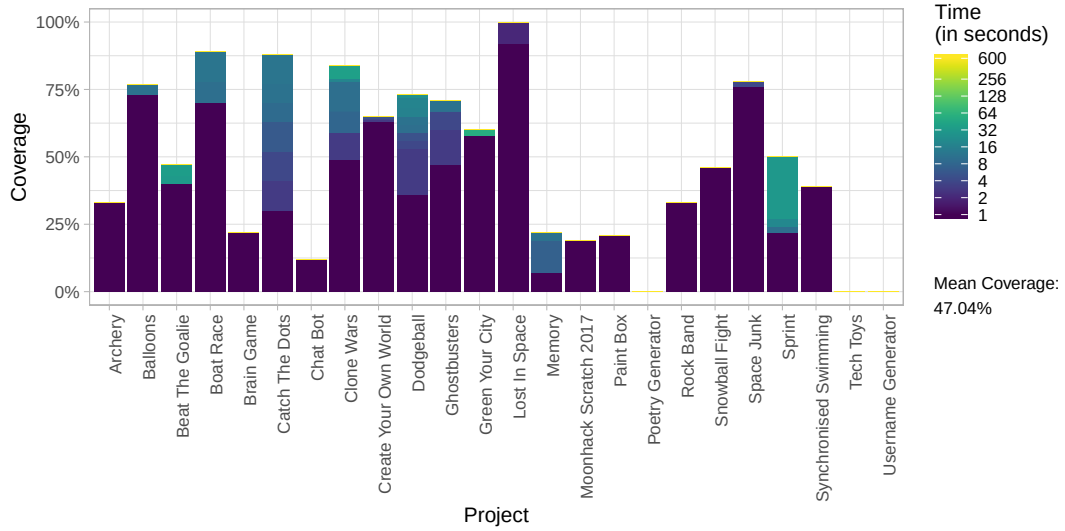


(a) Coverage per project

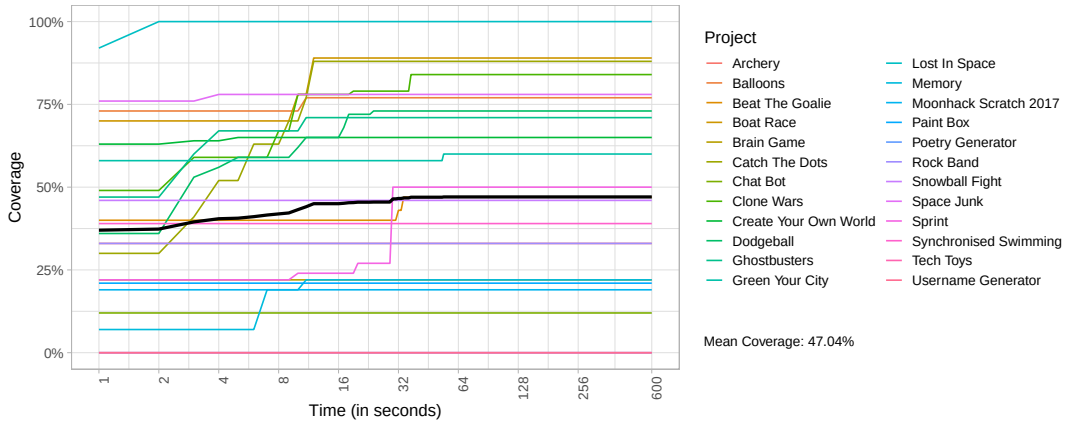


(b) Coverage over time

Figure 6.20: Achieved coverage with automated input averaged over ten runs

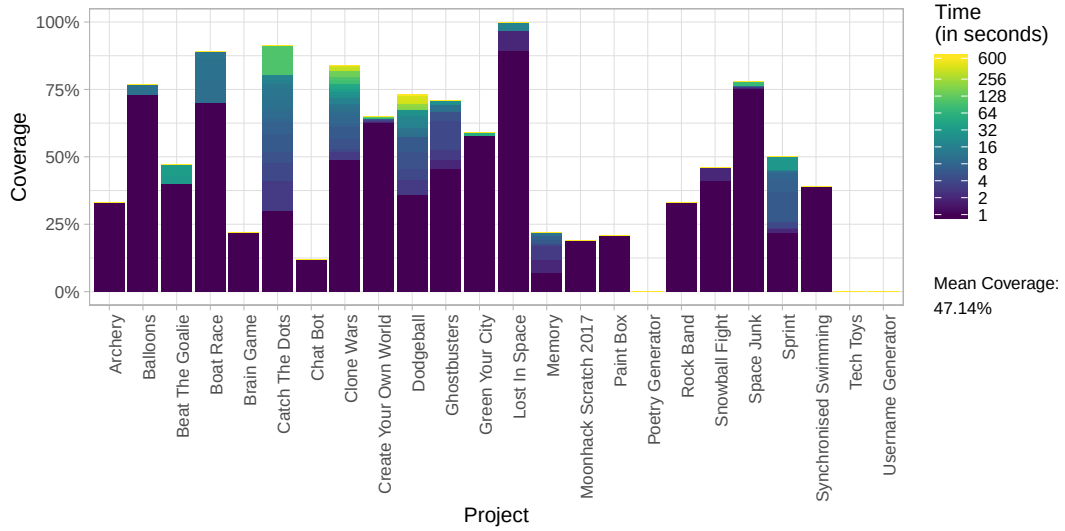


(a) Coverage per project

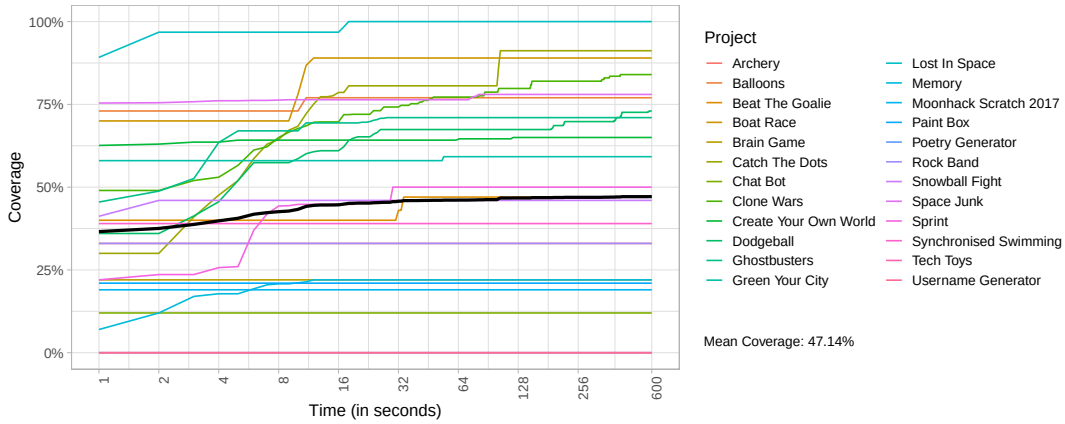


(b) Coverage over time

Figure 6.21: Achieved coverage without input on the first run



(a) Coverage per project



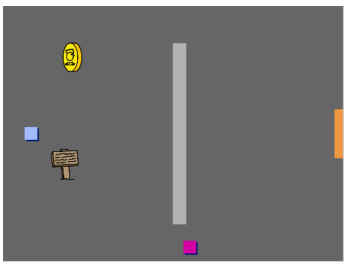
(b) Coverage over time

Figure 6.22: Achieved coverage without input averaged over ten runs

6.4.1 Reasons for Low Coverage on Certain Projects

Over the ten runs with generated input, we observed *Create Your Own World* and *Memory* consistently having the lowest coverage of the projects. We are going to take a quick look at the reasons, why these projects are difficult to cover using random input. A screenshot of both these programs can be seen in Figure 6.23.

- *Create Your Own World* implements a little adventure game with a player sprite that is moved using the arrow keys. The player starts on the left of the screen and is able to move through several rooms using a portal on the right side of the screen. This project is difficult to cover, because the sprite needs to move a long way in one direction to reach a screen transition. But since each arrow key has equal probability to be simulated by Whisker, the sprite is more likely to only hover over a small area around its start position.
- In *Memory* the player has to hit 4 different colored drums in an order that is randomly determined by the program. If the player hits one drum incorrectly, which includes hitting a drum while no order has been generated yet, the game goes into a game over state and can only be restarted by pressing the green flag. Therefore, using random inputs quickly results in an irrecoverable game over, because a wrong drum is hit, which renders much of the program's code unreachable.



(a) Create Your Own World

(b) Memory

Figure 6.23: Screenshots of difficult to cover Code Club projects

6.5 RQ4: Interference with Programs Under Test

In this section, we will answer the following research question:

RQ4: Does the testing process slow down the program under test?

Since JavaScript is executed in a single-threaded environment, Whisker executing code in between steps of the Scratch program can possibly delay execution steps of the program. This can be problematic because some of Scratch's blocks, specifically

the `wait` blocks, depend on real time. Therefore, executing additional code could influence time-related behaviour of the program, although we suspect that this will not be a concern for most Scratch programs.

Scratch's step function has to be called 30 times per second. Therefore, the execution time of Whisker's step procedure, which includes the execution of Scratch's step, should not exceed $1000\text{ms} \div 30 = 33.33\text{ms}$. Scratch allocates $0.75 \times 33.33\text{ms} = 25.0\text{ms}$ of the 33.33ms to execute the program. If no sprite changes occur during the step, the Scratch VM will execute the program for the allocated amount of time. But if some sprite's position or appearance changes, the VM will stop executing the program earlier. Afterwards, Scratch draws the new frame in the GUI. In the worst case, i.e. if no sprite changes and the whole time interval is used to execute the program, $0.25 * 33.33\text{ms} = 8.33\text{ms}$ will be left to render the new frame. Most modern hardware will take significantly less time than 8.33ms to render the picture, which leaves time for Whisker to execute other tasks.

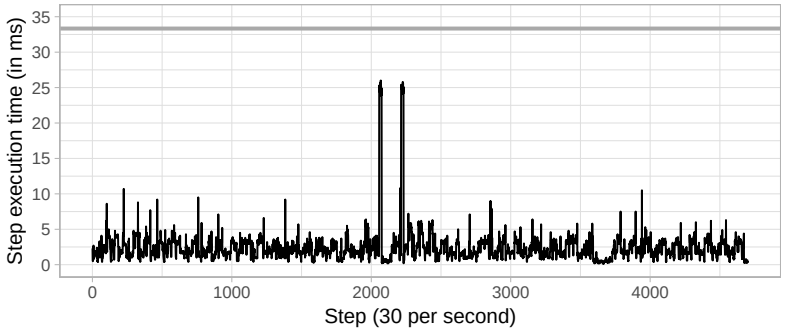
We should also note that coverage measurement gets done during Scratch's step, which will slow the program down if the full 25ms of execution time are used. However the Scratch GUI also registers code to update the user interface, which is run during the step of the VM, so this should not affect the program any more than Scratch's GUI itself does. Callbacks for sprite changes, i.e. `onSpriteMoved` and `onSpriteVisualChange`, are also executed during Scratch's step, but if one of these functions is executed, a redraw request will be issued anyways, causing the step execution to end earlier.

To evaluate this, we executed our test suites T1-T3 on the sample solution of the catching game (P1) ten times, and measured the execution time of Whisker's components (see Figure 5.3) as well as the execution time of Scratch during each step. Since we executed Whisker in a web browser, we were limited to JavaScript's `performance.now` method to measure execution times, which has an accuracy of 0.1ms. The code we used for measuring the execution times can be found in Listing A.6 in the appendix. We will consider the mean execution times, as well as the maximum execution times from the ten runs to decide if the tests could interfere with the program.

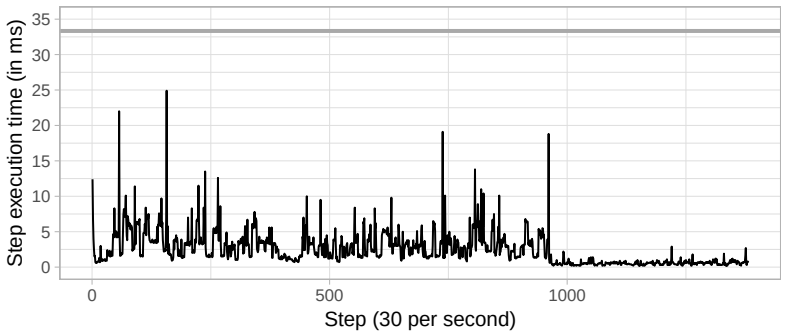
Null hypothesis (H_0): The execution of additional code delays the execution of Scratch's steps, which possibly interferes with the program execution.

Alternative hypothesis (H_1): The execution of additional code introduces no delay. It will therefore not interfere with the program execution.

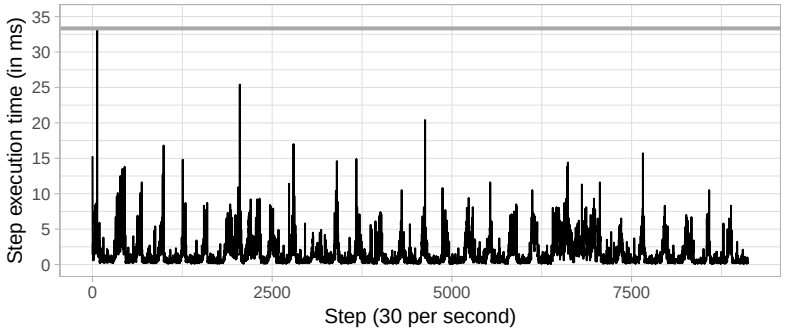
Figure 6.24 shows the total execution times from the first run of each test suite on the sample program. The time measurements include both the execution time of Whisker and Scratch. We can see that the execution time is mostly far beneath the limit of 33.33ms and only rarely spikes to higher for short periods of time.



(a) Time measurements for test suite T1 (normal)



(b) Time measurements for test suite T2 (constraint)



(c) Time measurements for test suite T3 (random input)

Figure 6.24: Measured execution times of Whisker's step procedure (in ms) during the first test execution

Table 6.25 shows the mean and the maximum of execution times for each part of Whisker's step procedure. This data is taken from ten consecutive executions of each test suite on the program under test. The average total step execution time is far beneath 33.33ms with 1.972ms, 2.272ms and 1.330ms for test suites T1-T3 respectively. The maximum total step execution times were 35.1ms, 24.9ms and 33.1ms. We can observe that the execution of test suite T1 spiked over the 33.33ms with a maximum of 35.1ms, but this only happened in one single step, with the second longest step execution time for T1 being 31.1ms.

	T1		T2		T3	
	mean	max	mean	max	mean	max
Callbacks (Before)	0.071	4.8	0.102	7.3	0.068	9.7
Random Inputs	0.004	1.3	0.004	3.1	0.024	6.4
Inputs	0.005	2.0	0.005	1.6	0.023	8.6
Sprites	0.022	6.4	0.022	4.2	0.022	4.8
Scratch (Program + Renderer)	1.852	35.1	2.054	24.6	1.129	32.7
Callbacks (After)	0.007	1.1	0.006	0.3	0.005	1.8
Constraints	0.012	2.9	0.078	3.2	0.058	12.5
Total	1.972	35.1	2.272	24.9	1.330	33.1
Total (Whisker only)	0.12	6.5	0.218	7.6	0.201	13.1

Table 6.25: Measured execution time (in ms) for every part of Whisker's step procedure over ten test executions

6.6 Discussion

In this evaluation, we conducted three separate experiments to examine the usefulness of Whisker. In the first experiment, we analyzed test results of three different test suites. We found out, that all of the results consistently show a strong correlation to independent manual scores. Therefore, we reject the null hypothesis of RQ1. We also proved that the constraint testing approach is able to deliver accurate test results, both with deliberate input and with automatically generated input.

We analyzed the flakiness of the test suites and measured a percentage of 4.15% inconsistent project-test combinations for test suite T1, 3.10% for test suite T2, and 6.32% for test suite T3. We found several reasons why certain tests and certain programs can be flaky. Many inconsistent results were caused by non-deterministic behaviour of the programs under test, which may be counteracted in the future by seeding Scratch's random number generator. At the same time, more experience in automated testing for Scratch will lead to more robust and less flaky test suites. The test suites we used were written early during development of Whisker. Therefore, the test suites had some more or less obvious flaws, some of which can be seen in Table 6.17 from the flakiness evaluation. And of course, tests that involve randomly

generated input show a higher number of inconsistent outcomes, since the tests themselves are non-deterministic.

The second experiment dealt with automated input generation. We let Whisker's automated input generation algorithm run on several Scratch programs ten times, then we ran the same programs without any input. We measured the achieved statement coverage on the programs during both executions, and compared their averages. The mean statement coverage amounted to 95.25% with automated input, and 47.14% without input. This shows that simple randomly generated input is able to reach much of typical Scratch programs' functionality.

Finally, we conducted another experiment to find out if the testing process would slow down the program under test. For this purpose, we measured the execution time of Whisker's step procedure on the execution of our test suites. We found that Scratch's allocated time interval for computations is long enough to allow executing both the Scratch program and test code. Sometimes, the execution time of the step procedure spiked over the allocated time interval, delaying a single step. However, these single spikes should not affect the programs in a meaningful way. The execution times will only be problematic if they consistently exceed the time limit. Therefore, programs under test were not affected by Whisker in our tests, and we can reject the null hypothesis of RQ4.

6.7 Threats to Validity

As is usual, the empirical studies we conducted in this work must confront certain threats to validity. Possible threats fall into one of two categories. On the one hand, threats to internal validity represent factors which provide alternative explanations for the presented results. On the other hand, threats to external validity deal with the generalizability of the results.

6.7.1 Internal validity

Conditions which could influence any test outcomes, correlations, coverage measurements, or time measurements are possible threads to internal validity.

Independence of ground truth. One threat to validity arises from the use of the manually assigned scores as ground truth to assess the quality of test results. If we assigned the scores ourselves, a comparison to them would not be adequate to give evidence about the quality of the results. However, the manual scores were assigned by the teacher after the course was held, and tests were written independently of the grading scheme, only using the textual specification of the program. Therefore, the manual scores are completely independent from our test results, and a comparison between the two is adequate.

Randomness. Another threat is, of course, random chance. Many Scratch programs use randomness, which can make test outcomes inconsistent. To eliminate the possibility of random chance affecting our test results as much as possible, we executed each test suite ten times, and considered the average results in addition to results from single executions.

RAM usage. One more possible threat comes from an issue of Scratch itself. At the time of the evaluation, Scratch had a problem of consuming increasing amounts of memory when loading many programs in succession, which, in theory, could have interfered with the execution of our test suites. However, we payed close attention to the RAM usage of the machine that executed the tests. We also performed some of our test suite executions on all projects sequentially, and some on one half of the programs, then later on the other half. We observed similar results with both methods, therefore this issue did not have any impact on our test results.

Excluded Projects. For RQ1, we excluded multiple Scratch projects from our statistics because they had various problems that would create a discrepancy between test results and manual scoring due to differences in the scoring methods. For the most part, these projects could not be started correctly, either because they did not use the green flag or because the projects were missing initialization and were saved in a state that causes an early game over. We conjecture that programs which were written with automated testing in mind won't have this kind of problems. If students are given a way to perform the tests on their program themselves, they will realize that the program is not starting correctly. And in most cases these problems are fairly easy to solve. Therefore, we believe that the statistics show more realistic results with these projects excluded.

6.7.2 External Validity

Factors which limit the generalizability of the results in this work are possible threats to external validity.

Programs under test. For one thing, the program under test could be chosen poorly, making the results not applicable for other Scratch programs. Therefore, we tried to choose programs that are representative of Whisker's target usage in the field of education. We chose Scratch projects from an actual workshop for students. The program has user interaction, uses randomness, and has a game over state, which are all important challenges for testing Scratch programs. To evaluate the automated input generation, we again went with programs from Scratch courses. The programs feature many different input methods and interactions, which make them suitable to evaluate automated input on.

Tests for time measurements. To find out if Whisker would slow down programs under test, we executed our test suites and measured the execution times

of Whisker's step procedure. One could argue that the execution times can be raised by a variety factors. More, as well as more computationally expensive, callbacks and constraints can be registered, more random inputs can be registered, and a Scratch program with more sprites can be chosen. Therefore, tests suites can interfere with the program under test, if they do it purposefully. However, we wanted to measure the execution time of a realistic test suite on a realistic program. This setup did not change the program's behaviour, and the measurements indicate that more expensive test suites will still be fine as well.

Chapter 7

Future Work

Although Whisker is usable in its current form, there are still many opportunities to expand upon in the future.

Automated input. While Whisker's automated input algorithm works quite well, it is still very simple. One could use more elaborate static analysis or search-based techniques to find better fitting inputs. For example, sequences of input may be found, and the optimal duration and probability of a key press or a sequence may be determined. At the same time, the algorithm may construct correct answers to `ask` blocks at run time. Figure 7.1 shows an ask-answer configuration from one of Code Club's sample solutions, for which answers may be generated.

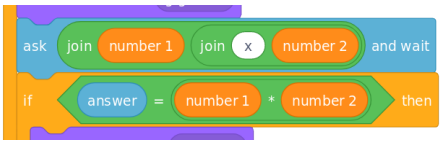


Figure 7.1: Ask-answer block configuration with a generated question and answer

Seeding randomness. Non-deterministic programs can be problematic for testing, since the behaviour of such programs may change from one execution to another. This is especially problematic for faulty implementations. Bugs may only sometimes cause incorrect behaviour, causing inconsistent test outcomes. This may be avoided by seeding the random number generator so it produces the same random numbers for each program execution.

Support for audio and Scratch extensions. Whisker could be extended to support audio as well as Scratch extensions. Scratch has a small number of extensions, which add blocks with various functionality. For example, the *Pen* extension allows users to freely draw on the stage by controlling a pen through the program, and the *Video Sensing* extensions allows users to detect movement with a web cam. Adding support for audio as well as extensions in the future would make Whisker applicable to a wider range of Scratch projects.

User interface. Currently, Whisker is only accessible through a web GUI, which is accessed through a web browser. This interface can be used to test projects in batch, but it still requires manual user interaction to select programs and tests, and to save the test report once the test execution finished. In the future, a standalone Electron[1] application could solve these problems by allowing Whisker to directly load projects and tests, and to directly save test reports. It would also make it possible to run tests in parallel more easily.

Simplify tests with helper methods. Tests that make use of Whisker can quickly become quite long and complex. Therefore, Whisker should should provide features to help simplify test code. For instance, one task that currently requires large amounts of code is checking temporal relationships between two events, for example "at most one second after some sprite touches a border, some variable must be increased". In the future, testing may be simplified greatly by providing methods to handle cases like this.

[1] `https://electronjs.org/`

Chapter 8

Conclusion

In this work, we took on the issue of automated assessment of Scratch programs. There exist multiple previous approaches that tackled this problem. We examined Hairball [2], which analyzes Scratch programs by performing static analysis on them, as well as ITCH [4], which transforms Scratch programs to automate `ask` and `say` blocks. We described various challenges, that have to be overcome in order to perform automated testing for Scratch. These challenges include Scratch's code system, which runs parallel scripts, its IO, which is not trivial to automate for testing, as well as some common traits in Scratch programs, which can make automated testing difficult.

As the main contribution of this work, we introduced Whisker, a utility that automates Scratch 3.0's IO, and described a way to perform runtime testing on Scratch programs by using this automation. Whisker allows tests to control the execution of Scratch programs, to programmatically simulate user input on them, and to obtain information about the sprites and variables of the program. It also offers additional features like measuring statement coverage of Scratch programs and automated test input generation.

We introduced a testing procedure that checks properties by defining constraints that the program under test must hold. Whisker checks these constraints in the background while the program is executed. This provides more flexibility about the execution of the program under test, making it possible to test with various sources of simulated user inputs, for example with randomly generated input. For this purpose, we implemented an automated input generation algorithm in Whisker, which detects what inputs the program can react to through static analysis, and randomly performs these inputs on the program.

We evaluated Whisker in three separate experiments. In the first experiment, we ran multiple test suites on a set of student-written Scratch programs. We learned that our testing approach is able to consistently produce test results that closely match the results of manual assessment. In the second experiment we tested the

automated input algorithm by measuring its achieved statement coverage on a set of different programs. We observed that it is able to cover most of usual Scratch programs' code, and produces much higher coverage compared to running the program without any simulated inputs. Finally, we investigated if the additional computations done by Whisker and the test code would interfere with the program under test by slowing it down, and discovered that this is not the case.

In conclusion, we proved automating Scratch's IO to be a viable way to perform functional testing on Scratch programs. We were able to achieve accurate test results both by alternatingly running the program and performing assertions, and by checking constraints in the background while the program is running. Therefore, we believe that automated testing may aid in the assessment of Scratch assignments in the future, both for grading purposes and for students or independent learners to check their own solutions.

Bibliography

[1] Bart, A. C., Tibau, J., Gusukuma, L., Tilevich, E., Shaffer, C. A., and Kafura, D. Blockpy. `https://think.cs.vt.edu/blockpy/`, 2013.

[2] Boe, B., Hill, C., Len, M., Dreschler, G., Conrad, P., and Franklin, D. Hairball: Lint-inspired static analysis of scratch projects. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2013), SIGCSE '13, ACM, pp. 215–220.

[3] Claessen, K., and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 46* (01 2000).

[4] Johnson, D. E. Itch: Individual testing of computer homework for scratch assignments. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (New York, NY, USA, 2016), SIGCSE '16, ACM, pp. 223–227.

[5] Keller, S. Using plan-driven development in educational programming projects, 2018.

[6] MIT Media Lab. Scratch. `https://scratch.mit.edu/`, 2009.

[7] Moreno-León, J., and Robles, G. Dr. scratch: A web tool to automatically evaluate scratch projects. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (New York, NY, USA, 2015), WiPSCE '15, ACM, pp. 132–133.

[8] Raspberry Pi Foundation. Code club. `https://codeclubprojects.org/`.

[9] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. Scratch: Programming for all. *Commun. ACM 52*, 11 (Nov. 2009), 60–67.

Statement of Authorship

I hereby declare that I am the sole author of this bachelor thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

(Place, Date) (Signature)

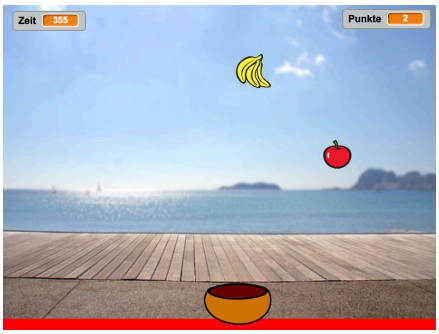
Appendix A

Additional Evaluation Data

Universität Passau	**Programmierkurs mit Scratch**	06.07.2018

THEMA: 3

ABSCHLUSSAUFGABE



Aufgabe – Früchte sammeln
`Dateiname: FS_deineNummer`

Benutze die bereits angelegte Scratch-Datei „FS“. Diese findest du unter „Meine Sachen“ in deinem Scratch-Account.

Spielbetrieb:

- Du musst mit der Schale („bowl“) Bananen und Äpfel sammeln.
- Ein Apfel bringt dir 5 Punkte, eine Banane 8 Punkte.
- Fällt der Apfel auf den Boden (rote Linie), ist das Spiel zu Ende.
- Fällt die Banane auf den Boden, gibt es 8 Punkte Abzug.
- Versuche möglichst viele Punkte zu sammeln, bis der Timer abgelaufen ist.

Details & Tipps für die Programmierung:

- Die Früchte fallen unterschiedlich schnell herunter. Die Änderung des y-Werts beim Apfel beträgt -5, bei der Banane -7.
- Wird die rote Linie vom Apfel berührt, erscheint am Apfel für eine Sekunde die Meldung „Game over!“ und das Spiel wird beendet.
- Wird die rote Linie von der Banane berührt, erscheint an der Banane für eine Sekunde die Meldung „-8“ und es gibt 8 Punkte Abzug.
- Der Timer läuft von 30 bis 0 im Sekundentakt rückwärts.
- Es dürfen nicht mehrere Früchte der gleichen Sorte gleichzeitig herunterfallen.
- Der Apfel und die Banane haben eine Größe von 50 %.
- Start der Früchte: x-Achse: zufällig / y-Achse: 170
- Die Früchte fallen gerade herunter.
- Sobald der Timer abgelaufen ist (auf 0 steht), erscheint an der Schale für eine Sekunde die Meldung „Ende!“ und anschließend wird das Spiel beendet.
- Sobald der Apfel oder die Banane die Schale berührt, wird der Punktestand geändert und er bzw. sie geht auf eine Zufallsposition mit y = 170 zurück und fällt erneut vom Himmel.
- Die Schale kann nur nach links bzw. rechts mit den entsprechenden Pfeiltasten im 10-er Schritt bewegt werden. Sie startet bei x = 0 / y = -145
- Die Banane soll nach dem Programmstart und auch nach der „-8“ Meldung (bei Berührung der roten Linie) eine Sekunde warten, bevor sie (wieder) vom Himmel fällt. Die Banane soll in dieser Wartezeit nicht sichtbar sein.

Figure A.1: Task description for the catching game (by Sebastian Keller [5])

#	Pts.	Description
1	2	Catching an apple gives 5 points, catching a banana gives 8 points.
2	2	The fruits fall down with different speeds. The y-coordinate of the apple is changed in steps of −5, the y-coordinate of the banana is changed in steps of −7.
3	3	When an apple touches the red line, a message saying "Game over!" appears on the apple for one second and the game ends.
4	3	When a banana touches the red line, a message saying "-8" appears on the banana for one second and 8 points are subtracted.
5	2	The timer ticks backwards from 30 to 0 in one-second intervals.
6	1	No two fruit of the same type must fall down at the same time.
7	2	The apple and the banana have a size of 50%.
8	2	Spawn point of the fruits: x-axis: random / y-axis: 170.
9	2	Fruits fall down in a straight line.
10	3	When the timer reaches 0, a message saying "Ende!" appears on the bowl for one second and the game ends afterwards.
11	3	When an apple or a banana touches the bowl, the score is changed, it is moved to a random position with y = 170, and it falls down again.
12	3	The bowl can only be moved left and right in steps of 10 with the respective arrow keys. It starts at x = 0 / y = -145.
13	4	The banana must wait one second before falling down when the program starts and after displaying the "-8" message. It must not be visible during that time.

Table A.2: Grading scheme for the manual scores (by Sebastian Keller [5], translated)

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
K6_S02	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0
K6_S03	0	0	0	0	0	0	0	**1**	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0
K6_S05	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	**1**	**1**	0	0	0
K6_S10	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S11	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	**1**	0
K6_S27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S31	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	0
K6_S33	0	0	0	**1**	0	0	0	**1**	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0
K7_S02	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0
K7_S03	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S04	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S05	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S06	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S08	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0
K7_S11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S12	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S14	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S16	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	**1**	0	**1**	**1**	0	0	0	0
K7_S19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table A.3: Inconsistency matrix for test suite T1 (normal)

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
K6_S02	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0
K6_S03	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	**1**	0	0	0
K6_S05	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	**1**	0	0	0
K6_S10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
K6_S11	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0
K7_S02	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S03	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S04	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S05	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S06	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S08	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0
K7_S10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0
K7_S11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0
K7_S14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0
K7_S15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0
K7_S16	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S17	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	**1**	0	0	0
K7_S19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table A.4: Inconsistency matrix for test suite T2 (constraint)

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
K6_S02	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0
K6_S03	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S05	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	0	**1**	0
K6_S10	0	0	0	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	0	**1**	0	0
K6_S11	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0
K6_S19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0
K6_S27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0	0
K6_S31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K6_S33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S02	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S03	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0	0
K7_S04	0	0	0	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S05	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0
K7_S06	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S08	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	0	**1**	0
K7_S10	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0
K7_S11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S12	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	**1**	**1**	0
K7_S14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0	0
K7_S15	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	**1**	**1**	0	0	0	0	**1**	0
K7_S16	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0
K7_S17	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	**1**	0
K7_S19	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	**1**	0	0	**1**	0	0
K7_S20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K7_S26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0

Table A.5: Inconsistency matrix for test suite T3 (random input)

```
const names = [
    'callbacks-before',
    'random-inputs',
    'inputs',
    'sprites',
    'scratch',
    'callbacks-after',
    'constraints'
];

let currentTimestamp;
let currentRecord;
let records;

const resetRecords = function () {
    records = [];
};

const startRecord = function () {
    currentTimestamp = window.performance.now();
    currentRecord = [];
};

const record = function () {
    currentRecord.push(window.performance.now() -
    ↪  currentTimestamp);
};

const endRecord = function () {
    records.push(currentRecord);
};

const getRecords = function () {
    return {
        names,
        records
    };
};
```

(a) Helper methods

```
step () {
    startRecord();

    this.callbacks.callCallbacks(false);
    record();

    if (!this.running) {
        endRecord();
        return;
    }

    this.randomInputs.performRandomInput();
    record();

    this.inputs.performInputs();
    record();

    this.sprites.update();
    record();

    this.vm.runtime._step();
    record();

    if (!this.running) {
        endRecord();
        return;
    }

    this.callbacks.callCallbacks(true);
    record();

    if (!this.running) {
        endRecord();
        return;
    }

    const rv = this.constraints.checkConstraints();
    record();

    endRecord();

    return rv;
}
```

(b) Modified step procedure

Listing A.6: Modified Whisker step procedure for measuring step execution times