

Een blokgebaseerd testframework voor Scratch

Iwijn Voeten

Studentennummer: 01800111

Promotoren: prof. dr. Peter Dawyndt, prof. dr. Christophe Scholliers

Begeleider: Niko Strijbol

Masterproef ingediend tot het behalen van de academische graad van

Master of Science in de informatica

Academiejaar 2022-2023

Samenvatting

Scratch is uitgegroeid tot een van de meest gebruikte tools om kinderen de kennis en vaardigheden van programmeren aan te leren. Scratch biedt een IDE en een blokgebaseerde programmeertaal aan in de browser. Het biedt een intuïtieve en visuele benadering van programmeren aan en laat toe om probleemoplossend denken bij kinderen aan te leren.

Scratch heeft geen ingebouwde testmogelijkheden. Hierdoor is het testen van Scratchprojecten beperkt tot de code uitvoeren en een visuele inspectie doen. Testen die automatisch feedback geven tijdens het leren programmeren kunnen helpen tijdens het leerproces. De bestaande tools om testen te schrijven laten niet toe om deze testen in Scratch zelf te schrijven. Dit verplicht lesgevers met kennis van Scratch om andere talen zoals JavaScript te leren. Bij tekstuele programmeertalen is het gangbaar om testen te schrijven in de programmeertaal die gecontroleerd wordt. Er ontbreekt dus nog een testframework dat het schrijven van testen voor Scratchprojecten mogelijk maakt in Scratch zelf. Daarom introduceert deze masterproef Poke: een testframework voor Scratch, in Scratch. Poke laat leerkrachten toe om in Scratch testen te schrijven voor Scratchprogramma's. De testen geven automatisch feedback geven aan kinderen tijdens het leerproces.

Poke werd geïmplementeerd als Scratchuitbreiding en voegt blokken toe aan de Scratchomgeving. Hierdoor kunnen bestaande Scratchblokken gebruikt worden tijdens het schrijven van testen. De blokken die Poke toevoegt maken het mogelijk om:

- Gebruikersinteractie te simuleren: druk op de groene vlag, beweeg de muis, druk op toetsen, klik op sprites.
- De omgeving te observeren: neem snapshots en haal daar later informatie uit waarmee controles mee kunnen gedaan worden.
- Feedback te geven: geef feedback aan de leerling.
- Scratchblokken uit te voeren in naam van een andere sprite: voer Scratchblokken uit in naam van een andere sprite om de testcode op 1 plaats te kunnen houden.

Poke werd gevalideerd door voor de Scratchoefeningen van de jeugdrondes van de Vlaamse Programmeerwedstrijd in 2017 testen te schrijven. Hieruit leerden we dat Poke nuttig kan gebruikt worden voor veel verschillende oefeningen.

A block-based test framework for Scratch

Iwijn Voeten, Niko Strijbol, Peter Dawyndt, Christophe Scholliers

Abstract

We present a block-based test framework for the Scratch programming environment named Poke. There are already several third party test frameworks for Scratch but none of them allows testing Scratch projects in Scratch itself. Poke is implemented as a Scratch extension and adds blocks that make it possible for teachers to write tests in Scratch that can provide automated feedback for students. Finally we validated the Poke extension by using it to test a set of exercises.

1. Introduction

Scratch (Resnick et al., 2009), a visual programming language widely used in educational settings, enables young learners to create interactive projects through a drag-and-drop interface: the Scratch GUI. The Scratch GUI is essentially the IDE of the Scratch programming language. It does not have a built in way of testing.

There are already several third party test frameworks available for Scratch, such as Whisker (Stahlbauer et al., 2019) and Itch (Mak et al., 2019). Both frameworks allow writing test plans in JavaScript. BASTET (Andreas et al., 2020) is a program analysis and verification framework for Scratch. BASTET converts Scratch code into an intermediate language and performs static testing on the code. Dr. Scratch (Moreno-León and Robles, 2015) is an analytical tool that evaluates Scratch projects. A more recent paper (Götz et al., 2022) introduces model-based testing of Scratch programs, where writing a test involves constructing a finite-state automaton (FSA) that describes the desired behavior of the program. Snap (Harvey et al., 2013), another block-based programming language, also has a test framework called SnapCheck (Wang et al., 2021), which allows writing tests in JavaScript.

The above testing frameworks usually require knowledge of JavaScript and don't allow writing tests in Scratch itself. To address this gap, we have developed Poke: a block-based test framework for Scratch. Poke leverages the block-based nature of Scratch to provide a user-friendly and intuitive testing environment, empowering teachers to write test cases directly within the Scratch interface. These tests can then automatically give feedback to students during their learning experience.

The research of this master's thesis has two objectives:

1. Investigate the possibility of a test framework for Scratch where the test code is written in Scratch, while it is common in textual programming languages to write test code in the same language as the code being tested. Such a framework does not yet exist for Scratch.
2. Explore what additional Scratch blocks are needed to write test code as ergonomically as possible.

Our strategy is to develop a proof-of-concept implementation of a block-based test framework to put ideas for points 1 and 2 into practice, and refine the prototype based on the insights gained. This approach is not aimed at completeness; it does not seek to test all Scratch projects or explore all possible testing strategies.

The goal is to write and execute test code to evaluate program code and gain insights into how test code can be written as ergonomically as possible. We want to try different alternatives and understand their limitations. At this stage, we are not concerned with how testing is ultimately implemented to test multiple solutions. Generating clear feedback is not a primary objective at the moment. We want to end up with a testing framework that can test a large body of exercises easily in Scratch.

The implementation is done through a Scratch extension called Poke. Some functionality that could not be added through the Poke extension was directly integrated into the

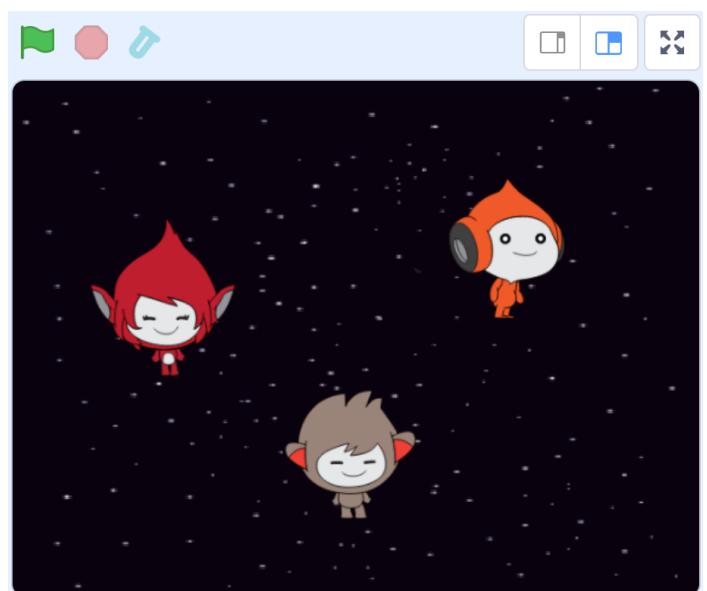


Figure 1: The 3 sprites "Giga", "Nano" and "Pico".

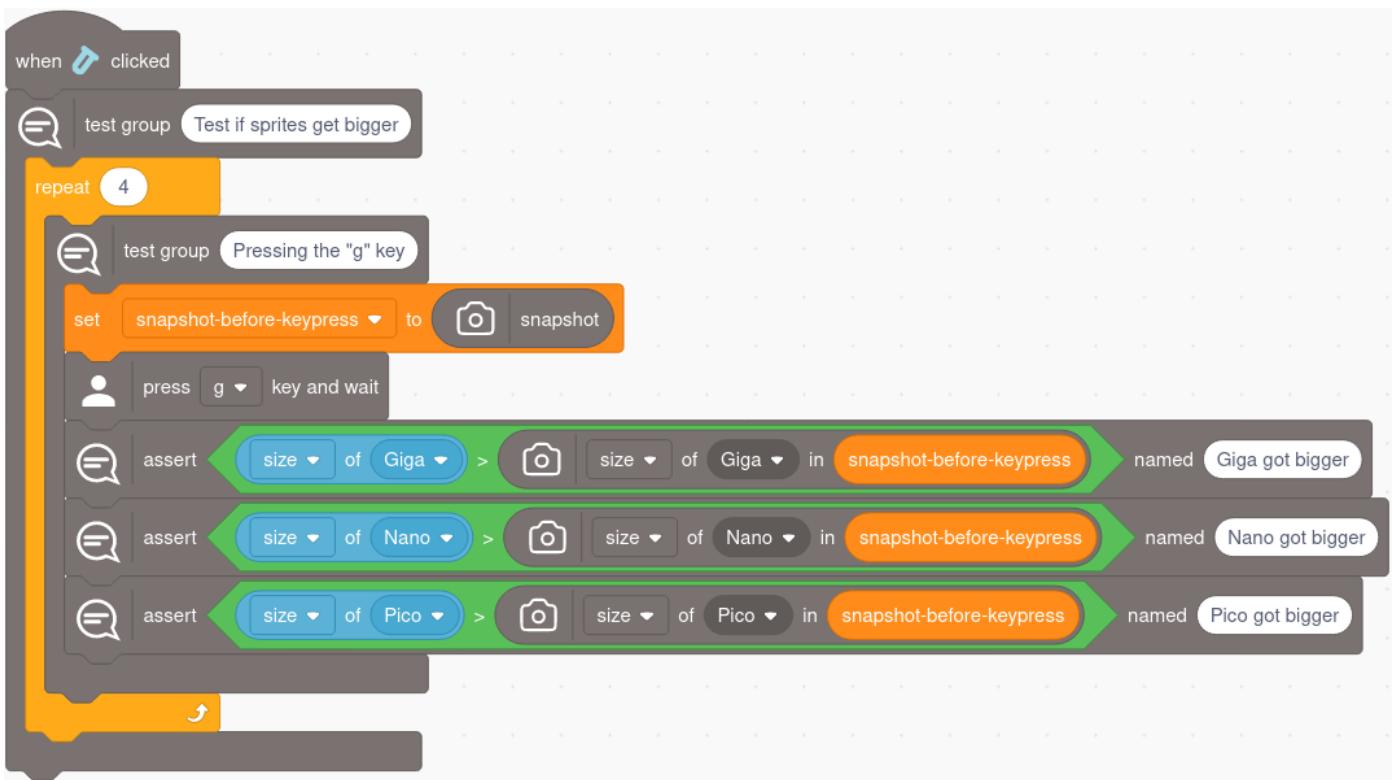


Figure 2: A test that checks whether the sprites "Giga", "Nano" and "Pico" get larger when pressing the g-key.

Scratch code base. We also aim to align with Scratch's philosophy as much as possible and create blocks that are intuitive for experienced Scratch programmers.

2. Poke

To give a better overview of what Poke does, we use a simple example exercise and show a part of the test code. The exercise consists of 3 sprites: "Giga", "Nano" and "Pico" (Figure 1). When the g-key is pressed, the sprites should increase in size, when the k-key is pressed their size should decrease. Solving this exercise is quite easy. Each sprite needs code that grows the sprite when the g-key is pressed, and shrinks the sprite when the k-key is pressed (Figure 3). We test our model solution by repeatedly:

- taking a snapshot of the environment, thereby saving the properties of each sprite, such as their size
- simulating a key press and waiting until the code of our model solution has handled this event. In this example this simply comes down to executing a *change size by* block
- polling the environment and comparing it to the snapshot to see if the sprites got bigger

The Poke extension consists of many other blocks. Table 2 shows an overview. Poke has blocks that provide user interaction simulation. Clicking on the green flag and on sprites is possible. Pressing keys and answering questions is too. The



Figure 3: In our model solution, this Scratch code is present in each sprite. It grows the sprite when the g-key is pressed, and shrinks it when the k-key is pressed.

feedback blocks give us the ability to construct feedback based on test results. The sensing blocks allow taking snapshots and their subsequent querying for properties. And finally the EBOS blocks allow us to execute Scratch blocks in other sprites to keep the test code centralized and avoid it being distributed over many different sprites.

Figure 2 shows Scratch code that makes use of the Poke extension (grey blocks) to test if the sprites grew when the g-key was pressed. This code will build a feedback tree. Figure 4 shows an example of a feedback tree where the test code noticed that, on one occasion, the sprite "Pico" did not get smaller when pressing the k-key. Now let's go through the test code snippet from Figure 2. It starts off with a hat-block that is activated when the blue test tube is pressed. The blue test tube is an extra button we added to the Scratch GUI and can be seen in the top right of Figure 1. Then we create a test group. A test group allows a teacher to group feedback together by creating an internal node in the feedback tree. Then we will press the g-key 4 times,

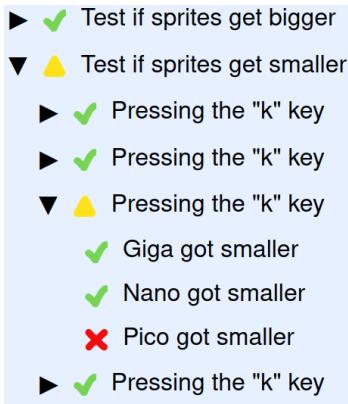


Figure 4: A feedback tree where, on one occasion, the sprite "Pico" did not get smaller when pressing the k-key.

each time also creating a test group. In every iteration we will use the *snapshot* block to so save its value in a Scratch variable "snapshot-before-key-press". Then we use the *press key and wait* block to simulate pressing the g-key. This block will wait for all Scratch code that started executing as a result of the key press to finish before continuing to the 3 *assert* blocks. The asserts blocks will simply compare the size of each sprite with the sizes when the snapshot was taken. If the condition of an *assert* block evaluates to 'true' it will add the feedback with a green check mark to the feedback tree, otherwise the feedback is added with a red cross.

3. Evaluation

As a validation for Poke we wrote tests for Scratch challenges that were used in the youth rounds of the Flemish Programming competition in 2017. These were also the challenges that the Itch thesis (Mak et al., 2019) used for validation. The challenges were divided into different categories, an overview can be seen in Table 1. There are 5 categories. The category "drawing" was the only category that could not be tested. This was because the Pen extension was used which Poke currently does not support. The category "talking" could mostly be tested. The main problem here being the inability to differentiate between what sprites are saying and what they are asking. The remaining categories "moving sprites", "animating sprites" and "games" could be tested well.

4. Conclusion

We were able to make a test framework for Scratch where the test code is written in Scratch itself. This was achieved by developing a Scratch extension called Poke. During the development we also explored what additional Scratch blocks were needed to write test code. We developed a proof-of-concept implementation of a block-based test framework and refined the prototype based on the insights gained. Poke is not a complete test framework that can test all Scratch projects. Generating clear feedback was also not a primary objective. For now feedback is provided through a simple tree structure leaving room for improvement.

We were able to write and execute test code to evaluate program code and gained insights into how test code can be written as ergonomically as possible. Finally, the Poke extension was validated by using it to test a set of example exercises showing Poke can be used to test a wide range, but not all, exercises.

References

- Andreas, S., Christoph, F., Gordon, F., 2020. Verified from scratch: Program analysis for learners' programs, in: ASE, IEEE.
- Götz, K., Feldmeier, P., Fraser, G., 2022. Model-based testing of scratch programs. URL: <https://arxiv.org/abs/2202.06271>, doi:10.48550/ARXIV.2202.06271.
- Harvey, B., Garcia, D.D., Barnes, T., Titterton, N., Armendariz, D., Segars, L., Lemon, E., Morris, S., Paley, J., 2013. Snap!(build your own blocks), in: Proceeding of the 44th ACM technical symposium on Computer science education, pp. 759–759.
- Mak, N., Dawyndt, P.p., Scholliers, C.c., 2019. Itch: een educatief test-framework voor automatische feedback op scratch projecten. URL: <http://lib.ugent.be/catalog/rug01:002787413>.
- Moreno-León, J., Robles, G., 2015. Dr. scratch: A web tool to automatically evaluate scratch projects, in: Proceedings of the Workshop in Primary and Secondary Computing Education, Association for Computing Machinery, New York, NY, USA. p. 132–133. URL: <https://doi.org/10.1145/2818314.2818338>, doi:10.1145/2818314.2818338.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al., 2009. Scratch: programming for all. Communications of the ACM 52, 60–67.
- Stahlbauer, A., Kreis, M., Fraser, G., 2019. Testing scratch programs automatically, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 165–175. URL: <https://doi.org/10.1145/3338906.3338910>, doi:10.1145/3338906.3338910.

	Type of exercise	Testable with Poke
Drawing	Drawing with the Pen extension	No, the Pen extension is currently not supported.
Talking	Sprites that talk, think or ask	Mostly, Poke can check what sprites are saying/thinking. Poke can also answer to asked questions. Poke can currently not discern between a sprite asking something or saying something.
Moving sprites	Sprites are moving around.	Yes, Poke can test the location of sprites.
Animating sprites	Sprites are using costume changes to show animations.	Yes, Poke can test what costume a sprite is wearing.
Games	Minigames that require constant user interaction.	Yes, Poke is able to simulate the user interaction and query the environment for testing.

Table 1: The different categories of the validation exercises.

	Capabilities of Poke	Blocks
User interaction simulation	Clicking the green flag	
	Clicking sprites	
	Moving the mouse	
	Pressing keys	
	Answering questions	
Feedback	Giving correct feedback	
	Giving wrong feedback	
	Giving feedback based on a condition	
	Making a test group	
	Waiting until a condition is true for a maximum of x seconds. If the timeout was reached, execute the second branch, otherwise execute the first.	
Sensing	Taking snapshots	
	Getting information from snapshots such as: sprite size, position, what they are saying and thinking, ...	
Execution of Blocks in Other Sprites (EBOS)	Execute Scratch code in other sprites	
	Grouping all sprites that adhere to a condition in a list	

Table 2: The blocks available through the Poke extension

Wang, W., Zhang, C., Stahlbauer, A., Fraser, G., Price, T., 2021. Snapcheck: Automated testing for snap! programs, in: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1, Association for Computing Machinery, New York, NY, USA. p. 227–233. URL: <https://doi.org/10.1145/3430665.3456367>, doi:10.1145/3430665.3456367.

Dankwoord

Ik bedank mijn promotoren prof. dr. Peter Dawyndt en prof. dr. Christophe Scholliers en mijn begeleider Niko Strijbol voor alle hulp. De begeleiding die zij boden in de vorm van wekelijkse meetings en constante feedback was fantastisch. Ik ken geen enkele medestudent die een betere thesis begeleiding heeft gehad dan mij.

Daarnaast bedank ik een medestudent en enorm goede vriend Klaas Goethals. Hij deed ook zijn thesis rond Scratch waardoor we elkaar vaak hulp konden aanbieden en gaf me daarbovenop ook morele steun.

Tenslotte wil ik mijn ouders Bruno Voeten en Bianca Vermeersch en mijn vriendin Jade Hazenberg ook nog bedanken voor hun morele steun en motivatie.

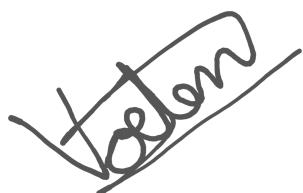
Iwijn Voeten
25/05/2023

Toelating tot bruikleen

De auteur geeft de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de bepalingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.

Iwijn Voeten

25/05/2023

A handwritten signature in black ink, appearing to read "Iwijn Voeten". The signature is written in a cursive style and is oriented diagonally from the bottom-left towards the top-right.

Inhoudsopgave

1 Inleiding	1
1.1 Wat is Scratch?	1
1.1.1 Testcode	3
1.2 Het doel	3
1.3 Bestaande technologieën	5
1.3.1 Dr. Scratch	5
1.3.2 Itch	5
1.3.3 Whisker	7
1.3.4 SnapCheck	11
1.4 Conclusie	12
2 Poke	13
2.1 Een eenvoudig voorbeeld	14
2.2 Feedbackblokken	15
2.2.1 Een blad toevoegen aan de feedbackboom	15
2.2.2 Interne toppen toevoegen aan de feedbackboom	16
2.2.3 Wachten tot iets gebeurt	17
2.3 Gebruikersinteractieblokken	18
2.3.1 Threads	18
2.3.2 <i>and wait</i> blokken	18
2.3.3 Press green flag	19
2.3.4 Press key	19
2.3.5 Press key and wait	19
2.3.6 Move mouse to	20
2.3.7 Click sprite	20
2.3.8 Click sprite and wait	20
2.3.9 Answer	21
2.4 Waarnemingsblokken	21
2.4.1 Snapshots nemen	21
2.4.2 Informatie uit snapshots halen	21
2.5 Uitvoeren van Blokken in een Andere Sprite (UBAS)	22
2.5.1 With sprite do	23
2.5.2 Sprites selecteren met een filter	23
2.6 Tips bij het schrijven van testen	24
2.6.1 Testcode in een aparte sprite	24
2.6.2 Dezelfde code uitvoeren in meerdere sprites	24

2.6.3	Interferentie vermijden	24
2.6.4	Het <i>query snapshot</i> blok in combinatie met het <i>snapshot</i> blok gebruiken	26
2.7	Ontbrekende functionaliteit	26
2.7.1	Meerdere <i>when tests started</i> blokken	26
2.7.2	Statisch testen van Scratchblokken	26
2.7.3	Spritespecifieke editoropties bij UBAS	27
2.8	Conclusie	28
3	Een uitbreiding toevoegen aan Scratch	29
3.1	Architectuur	29
3.2	Uitbreidingsklasse	30
3.3	Een blok definiëren	32
3.3.1	Een startblok	33
3.3.2	Een C-blok	35
3.3.3	Verslaggeversblokken	36
3.3.4	Argument types	37
3.3.5	Menu's	38
3.4	Takken uitvoeren	39
3.5	Wachten	40
3.6	Conclusie	40
4	Poke: technisch bekeken	41
4.1	Uitvoeren van Blokken in een Andere Sprite (UBAS)	41
4.1.1	Blokinjectie	42
4.1.2	Het <i>when broadcast received</i> blok toevoegen	43
4.1.3	Het <i>when broadcast received</i> blok starten	43
4.1.4	UBAS efficiënt maken	44
4.1.5	De UBAS-blokken	45
4.1.6	Discussie	48
4.2	Feedback geven	48
4.2.1	Het <i>test group</i> blok	50
4.2.2	Het <i>assert</i> blok	51
4.2.3	Het <i>wait until or timeout</i> blok	51
4.2.4	Discussie	52
4.3	Gebruikersinteractie simuleren	52
4.3.1	Startblokken activeren en er op wachten	52
4.3.2	De muisaanwijzer bewegen	53
4.3.3	De vragenwachtlijst	54

4.4	Waarnemingen doen	55
4.5	Conclusie	55
5	Validatie	56
5.1	Op bezoek bij Devin	56
5.2	Flauw mopje	60
5.3	Cijfersom	64
5.4	Heksenjacht	67
5.5	Mad hatter	70
5.6	Voetballende kat	71
5.7	Vliegende papegaai	75
5.8	Vang de appels	78
5.9	Tekenen	82
5.10	Conclusie en toekomstige uitbreidingen	82
6	Conclusie	84
6.1	Toekomstig werk	84

1 Inleiding

Scratch (Resnick e.a. 2009) is uitgegroeid tot een van de meest gebruikte tools om kinderen de kennis en vaardigheden van programmeren aan te leren. Scratch biedt een IDE en een blokgebaseerde programmeertaal aan in de browser. Het biedt een intuïtieve en visuele benadering van programmeren aan en laat toe om probleemoplossend denken bij kinderen aan te leren. Scratch is het meest geschikt voor kinderen tussen 10 en 14 jaar, en kan zowel binnen en buiten school gebruikt worden om te leren programmeren.

Scratch heeft geen ingebouwde testmogelijkheden. Hierdoor is het testen van Scratchprojecten beperkt tot de code uitvoeren en een visuele inspectie doen. Softwaretesten die automatisch feedback geven tijdens het programmeren kunnen mogelijk helpen tijdens het leerproces. De bestaande tools om testen te schrijven laten niet toe om deze testen in Scratch zelf te schrijven. Dit verplicht lesgevers met kennis van Scratch om andere talen zoals JavaScript (Ecma 1999) te leren. Er ontbreekt dus nog een testframework dat het schrijven van testen voor Scratchprojecten mogelijk maakt in Scratch zelf. Terwijl het voor tekstuele programmeertalen wel gebruikelijk is om testcode te schrijven in dezelfde programmeertaal als de code die moet getest worden.

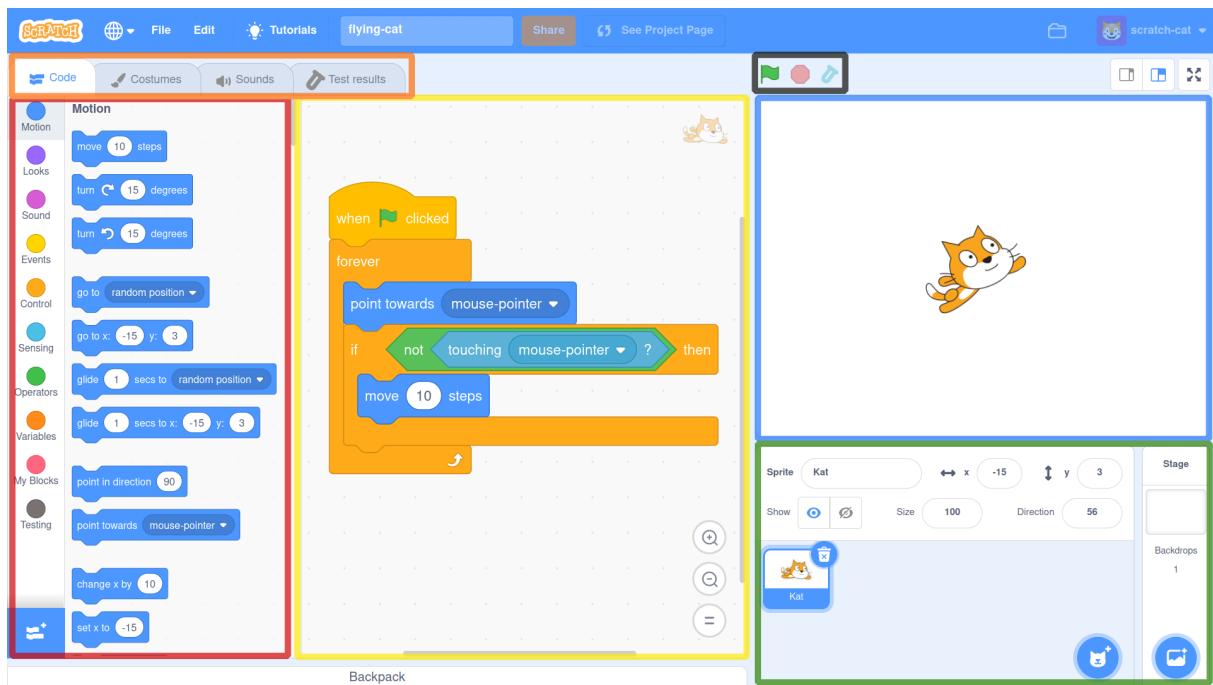
1.1 Wat is Scratch?

Scratch is een visuele programmeertaal die ontwikkeld is door Lifelong Kindergarten group van het MIT Media Lab (Resnick e.a. 2009). Het is ontworpen om kinderen op een creatieve en intuïtieve manier kennis te laten maken met programmeren. In Scratch kunnen gebruikers programma's maken door blokken met instructies te slepen en neer te zetten. Deze blokken zijn grafische weergaven van programmeerconstructies zoals lussen, voorwaardelijke instructies, variabelen en tal van andere blokken die de Scratchomgeving kunnen beïnvloeden. Met Scratch kunnen gebruikers interactieve verhalen, spelletjes en andere creaties maken.

We bespreken nu een voorbeeldoefening om een aantal basisconcepten binnen Scratch uit te leggen. Het doel van deze oefening is om de kat richting de muisaanwijzer te laten vliegen. De Scratchomgeving is te zien in Figuur 1. Daar zijn een aantal gekleurde kaders te zien:

- Geel: de Scratchcode. Dit is de voorbeeldoplossing van de oefening “de vliegende kat”.
- Rood: het selectie menu waar alle blokken te zien zijn. De blokken zijn onderverdeeld in categorieën. De onderste categorie “testing” werd toegevoegd in deze masterproef.
- Blauw: het Scratchcanvas waar de sprite “Kat” kan bewegen.
- Groen: hier staan alle sprites (in dit voorbeeld maar 1). Klikken op een sprite zal in het gele vak de Scratchcode die bij de sprite hoort tonen. Momenteel krijgen we dus de Scratchcode te zien van de sprite “Kat”.

- Oranje: de verschillende tabbladen. Het tabblad “test results” werd toegevoegd in deze masterproef.
- Zwart: een aantal knoppen. De groene vlag zal hier het programma starten. De rode “stop”-knop zal alle Scratchcode stoppen, en blauwe testtube werd toegevoegd in deze masterproef om de testen te starten.



Figuur 1: De Scratchomgeving waar de oefening “de vliegende kat” in te zien is.

We zien dat de code begint uit te voeren als op de groene vlag gedrukt wordt. Daarna zal in een oneindige lus code uitgevoerd worden. In deze code wordt de sprite telkens gericht naar de muisaanwijzer, waarna de kat 10 stappen zal zetten als hij de muisaanwijzer nog niet aanraakt.

De voorbeeldoplossing bestaat uit één stapel blokken. Een uitvoering van zo een stapel noemt een thread. Een sprite kan meerdere threads hebben die gelijktijdig uitvoeren.

De voorbeeldoplossing bevat de volgende soorten blokken:

- Startblok: *when green flag pressed* is een startblok. Startblokken dienen om threads te starten.
- C-blok: het *forever* blok en het *if* blok zijn C-blokken. C-blokken hebben een C-vorm waar andere C-blokken of commandoblokken in geplaatst kunnen worden.
- Commandoblok: het *move steps* blok en het *point towards* blok zijn commandoblokken. Commandoblokken zijn simple instructies en geven geen waarde terug.
- verslaggeversblokken: het *not* en *touching ?* blok zijn verslaggeversblokken. Verslaggeversblokken geven een waarde terug die kan gebruikt worden door andere blokken. Deze 2 verslagge-

versblokken hebben een zes-hoekige vorm. Dat betekent dat ze een Booleaanse waarde teruggeven. Verslaggeversblokken kunnen ook een afgeronde vorm hebben. Dan geven ze een string terug.

1.1.1 Testcode

De Scratchcode die de kinderen schrijven om een project te implementeren zoals besproken in Paragraaf 1.1 noemen we in deze masterproef de programmacode. Dit is de code die een leerling schrijft en dus moet getest worden. De Scratchcode die gebruikt wordt om programmacode te testen noemen we testcode.

1.2 Het doel

Het onderzoek van deze masterproef is tweevoudig:

1. Onderzoeken of er een testframework voor Scratch kan ontwikkeld worden waarvan de testcode geschreven wordt met Scratchblokken. Hier wordt verondersteld dat zo een testframework nog niet bestaat, terwijl het voor tekstuele programmeertalen wel gebruikelijk is om testcode te schrijven in dezelfde programmeertaal als de programmacode die moet getest worden.
2. Onderzoeken welke extra Scratchblokken nodig zijn om zo ergonomisch mogelijk testcode te kunnen schrijven.

De strategie is om een proof-of-concept implementatie van een blokgebaseerd testframework uit te werken waarmee we ideeën voor punt 1 en 2 in de praktijk kunnen brengen, en op basis daarvan het prototype verder te verfijnen met de bekomen inzichten. Met deze strategie doen we niet op volledigheid, we willen niet per se alle Scratchprojecten kunnen testen of alle mogelijke teststrategieën onderzoeken.

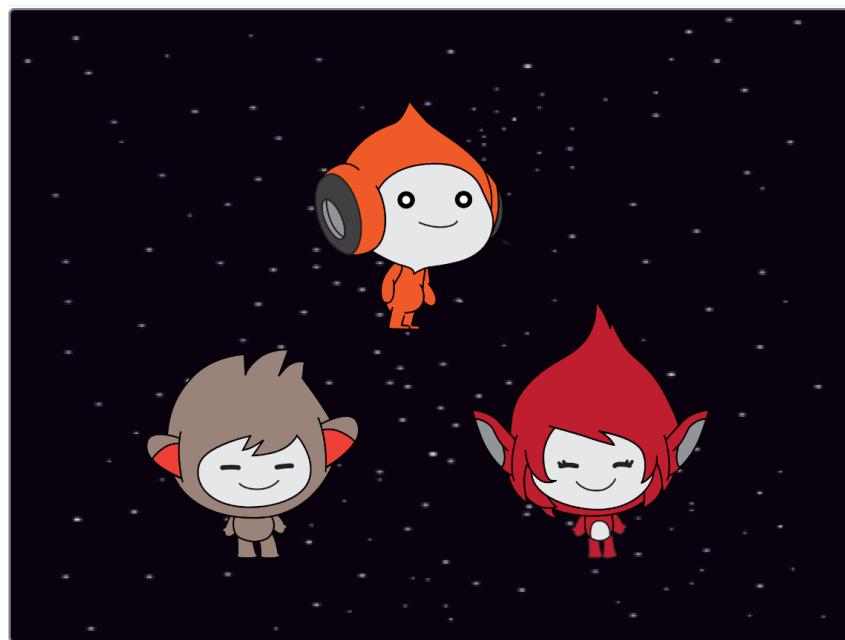
Het doel is dat we zelf testcode kunnen schrijven en uitvoeren om programmacode te beoordelen en zo inzichten te verwerven over hoe testcode zo ergonomisch mogelijk kan geschreven worden. We willen verschillende alternatieven uitproberen en beperkingen leren kennen. Hierbij willen we ons niet bekommernen over hoe het testen final in de praktijk gebracht wordt om meerdere oplossingen te testen. Genereren van zo duidelijk mogelijke feedback is op dit moment ook geen hoofddoel.

De implementatie gebeurt via een Scratchuitbreiding genaamd Poke. Sommige functionaliteit kon niet via de Poke-uitbreiding toegevoegd worden en werd rechtstreeks in de Scratch codebase toegevoegd.

Poke kan gebruikt worden door leerkrachten om testen te schrijven die automatische feedback geven aan kinderen tijdens het programmeren. Poke voegt blokken toe aan Scratch waarmee testplannen

opgesteld kunnen worden die automatisch feedback genereren. We proberen hierbij zoveel mogelijk aan te leunen bij de filosofie van Scratch en doelen op zo intuïtief mogelijke blokken voor ervaren Scratchprogrammeurs.

Om de eerste testblokken te ontwerpen werd gekeken naar welke testblokken er nodig waren om de oefening “groeien en krimpen” te kunnen testen. In deze oefening zijn er 3 sprites (zie Figuur 2). Als op de g-toets wordt gedrukt moeten de sprites groter worden. Als op de k-toets wordt gedrukt moeten de sprites kleiner worden.



Figuur 2: Het canvas van de oefening “groeien en krimpen”.

Een voorbeeldoplossing is te zien in Figuur 3. Daar zien we de Scratchcode die aanwezig is in elke sprite. Deze programmacode vangt het indrukken van de g-toets en k-toets op en maakt de sprite respectievelijk groter en kleiner.



Figuur 3: Een voorbeeldoplossing van de *groeien en krimpen* oefening. Elke sprite bevat deze Scratchcode.

De oefening “groeien en krimpen” wordt gebruikt om een aantal bestaande technologieën te beoordelen.

1.3 Bestaande technologieën

Er bestaan al testframeworks voor Scratch: Whisker (Stahlbauer, Kreis, en Fraser 2019) en Itch (Mak, Dawyndt, en Scholliers 2019) zijn daar voorbeelden van. Beide geven de mogelijkheid om testplannen te schrijven in JavaScript, maar laten niet toe om testplannen in Scratch zelf te schrijven.

BASTET (Andreas, Christoph, en Gordon 2020) is een programma-analyse- en verificatieframework voor Scratch. BASTET zet Scratch om naar een tussentaal en voert statische testen uit op de code. Dr. Scratch (Moreno-León en Robles 2015) is een generieke analytische tool die Scratchprojecten evalueert zonder specifiek testplan.

Een recentere paper (Götz, Feldmeier, en Fraser 2022) introduceert modelgebaseerd testen van Scratchprogramma's, waarbij het schrijven van een testplan inhoudt om een eindigetoestandsautomaat (FSA) op te stellen die het gewenste gedrag van het programma beschrijft.

Ook voor Snap, een andere blokgebaseerde programmeertaal, bestaat een testframework genaamd SnapCheck (Wang e.a. 2021b) waarmee testen in JavaScript kunnen geschreven worden.

1.3.1 Dr. Scratch

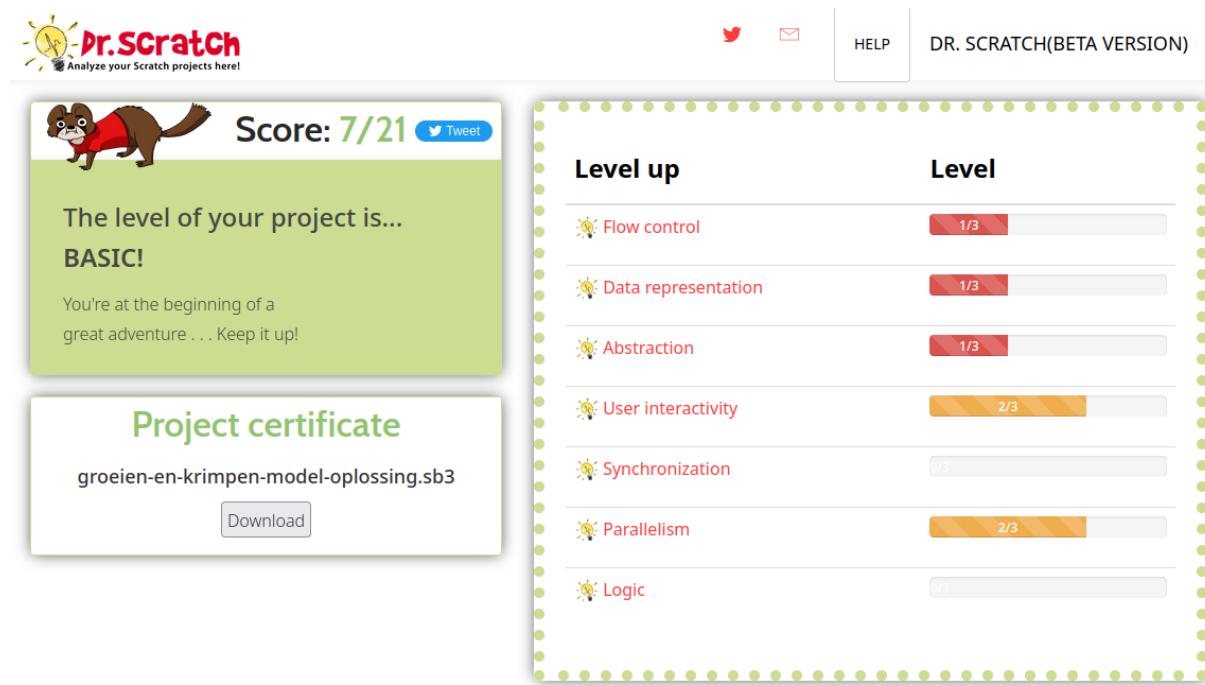
Dr. Scratch (Moreno-León en Robles 2015) is een tool die statisch naar de gebruikte blokken in een Scratchproject gaat kijken en een score geeft. Figuur 4 toont de evaluatie van onze modeloplossing voor “groeien en krimpen”.

Een voordeel van Dr. Scratch is dat er geen testplan nodig is. De nadelen zijn dat de score afhangt van generieke criteria die zij “code smells” noemen. De voorbeeldoplossing van “groeien en krimpen” gebruikt bijvoorbeeld geen herhalingen omdat dit niet nodig is, maar hierdoor geeft Dr. Scratch een lagere score.

Dr. Scratch voert een statische analyse uit op van een Scratchproject terwijl Poke toelaat om oefeningsspecifieke testplannen op te stellen die gericht feedback geven aan kinderen.

1.3.2 Itch

Itch laat toe om testen te schrijven in JavaScript. Onderstaand codefragment stelt een test voor die 4 keer de g-toets indrukt en telkens controleert of de sprites groter worden. Daarna wordt 4 keer op de k-toets gedrukt en wordt telkens gecontroleerd of de sprites kleiner worden.



Figuur 4: De evaluatie van een modeloplossing voor de oefening “groeien en krimpen”.

```

1 function duringExecution(e) {
2     e.actionTimeout = 10000;
3     e.turboMode = true;
4     e.acceleration = 5;
5     e.eventAcceleration = 1;
6     const spriteToOldSizeMap = {
7         "Giga": e.vm.runtime.getSpriteTargetByName("Giga").size,
8         "Pico": e.vm.runtime.getSpriteTargetByName("Pico").size,
9         "Nano": e.vm.runtime.getSpriteTargetByName("Nano").size
10    }
11    e.scheduler
12      .forEach([...Array(4).keys()], (prev) => {
13        return prev.pressKey('g').log(() => {
14          e.out.group("Test if sprites get bigger", () => {
15            for (const spriteName in spriteToOldSizeMap) {
16              const currentSpriteSize = e.vm.runtime.
17                getSpriteTargetByName(spriteName).size;
18              e.out.test(`${spriteName} got bigger`)
19              .expect(currentSpriteSize > spriteToOldSizeMap[spriteName]
20                  ].toBe(true));
21              spriteToOldSizeMap[spriteName] = currentSpriteSize
22            }
23          });
24      });
25  }

```

```

24     .forEach([...Array(4).keys()], (prev) => {
25       return prev.pressKey('k').log(() => {
26         e.out.group("Test if sprites get smaller", () => {
27           for (const spriteName in spriteToOldSizeMap) {
28             const currentSpriteSize = e.vm.runtime.
29               getSpriteTargetByName(spriteName).size;
30             e.out.test(`#${spriteName} got smaller`)
31               .expect(currentSpriteSize < spriteToOldSizeMap[spriteName]
32                 ]).toBe(true);
33             spriteToOldSizeMap[spriteName] = currentSpriteSize
34           }
35         });
36       .end();
37     })
  
```

In Figuur 5 is dezelfde test te zien, maar dan geschreven in Scratch en gebruikmakend van de Poke-uitbreiding. De Itch test en Poke test zijn functioneel equivalent.

Itch laat toe om oefeningen zeer grondig te testen, maar door het gebruik van JavaScript is de instapdrempel hoog. Met de Poke-uitbreiding is het mogelijk om deze test met een beperkt aantal blokken in Scratch te maken.

In Hoofdstuk 2 wordt dieper ingegaan op de uitleg van de grijze blokken die te zien zijn in Figuur 5.

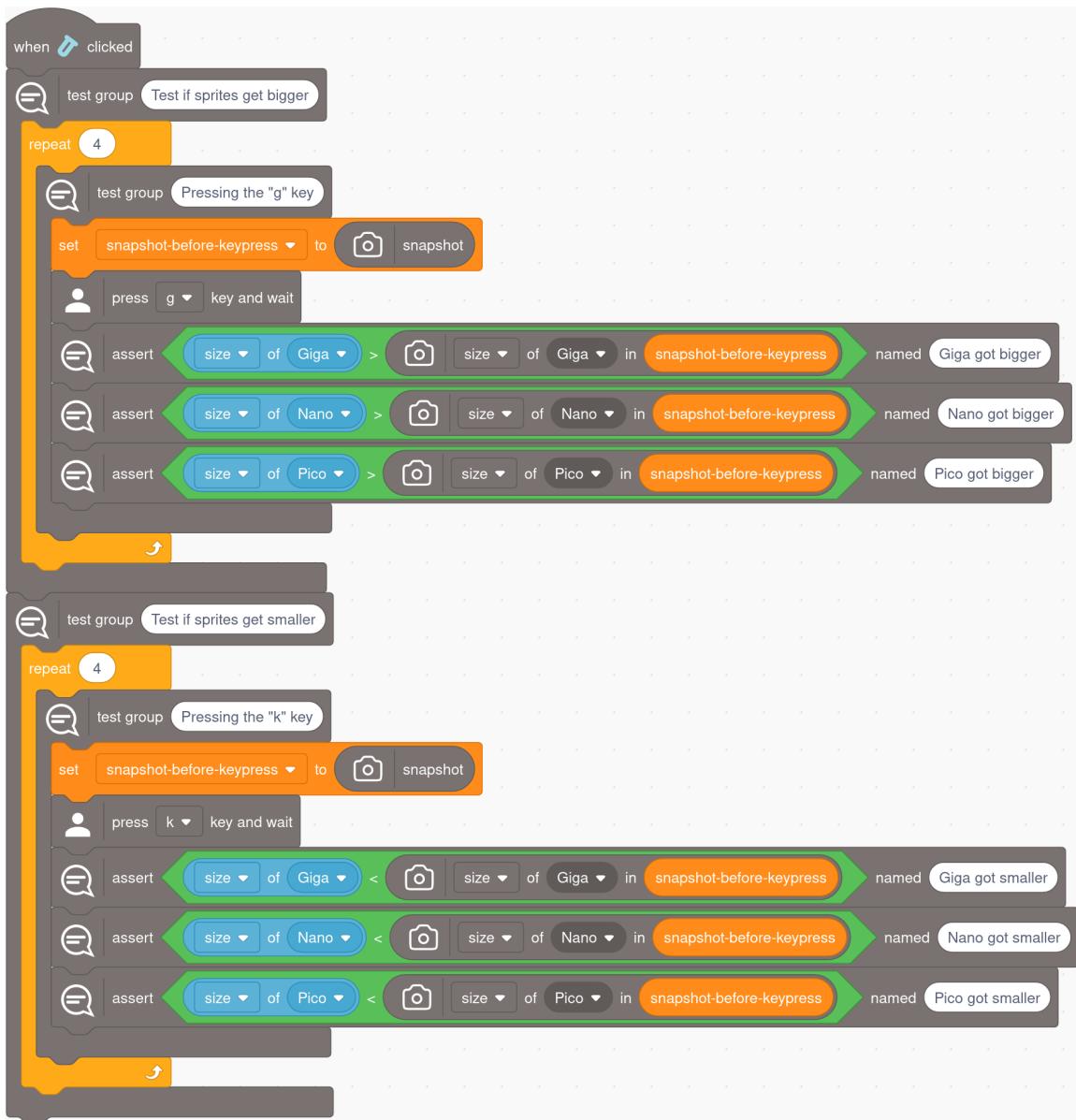
1.3.3 Whisker

Whisker is een testframework voor Scratch dat verschillende teststrategieën toelaat. In Whisker kan een Scratchproject geladen worden samen met de testen. Whisker kan ook testen genereren, of gebruikt worden als test editor.

Hier zien we dat Whisker op basis van een voorbeeldoplossing in Scratch voor de oefening “groenen en krimpen” 2 testen gegenereerd heeft in JavaScript:

```

1 const test0 = async function(t) {
2   t.keyPress('g', 1);
3   await t.runForSteps(1);
4   await t.runForSteps(1);
5 }
6 const test1 = async function(t) {
7   t.keyPress('g', 1);
8   await t.runForSteps(1);
9   await t.runForSteps(1);
10  t.keyPress('k', 1);
11  await t.runForSteps(1);
12  await t.runForSteps(1);
13 }
  
```



Figuur 5: De oefening “groeien en krimpen” getest met behulp van de Poke-uitbreiding.

```
14 module.exports = [
15     test: test0,
16     name: 'Generated Test',
17     description: '',
18     categories: [],
19     generationAlgorithm: 'mio',
20     seed: '1671449523126',
21     type: 'standard',
22 },
23 [
24     test: test1,
25     name: 'Generated Test',
26     description: '',
27     categories: [],
28     generationAlgorithm: 'mio',
29     seed: '1671449523126',
30     type: 'standard',
31 ]
32 ]
```

De eerste test drukt op de g-toets. De tweede test drukt eerst op de g-toets en dan op de k-toets. Whisker had door dat het Scratchprogramma moet reageren op het indrukken van de g-toets en de k-toets, maar is er niet in geslaagd om code te genereren die test of de sprites vergroten of verkleinen.

Na het genereren van testen kunnen de testen aangepast worden in Whisker. Ook kunnen de testen uitgevoerd worden om het Scratchproject te controleren.

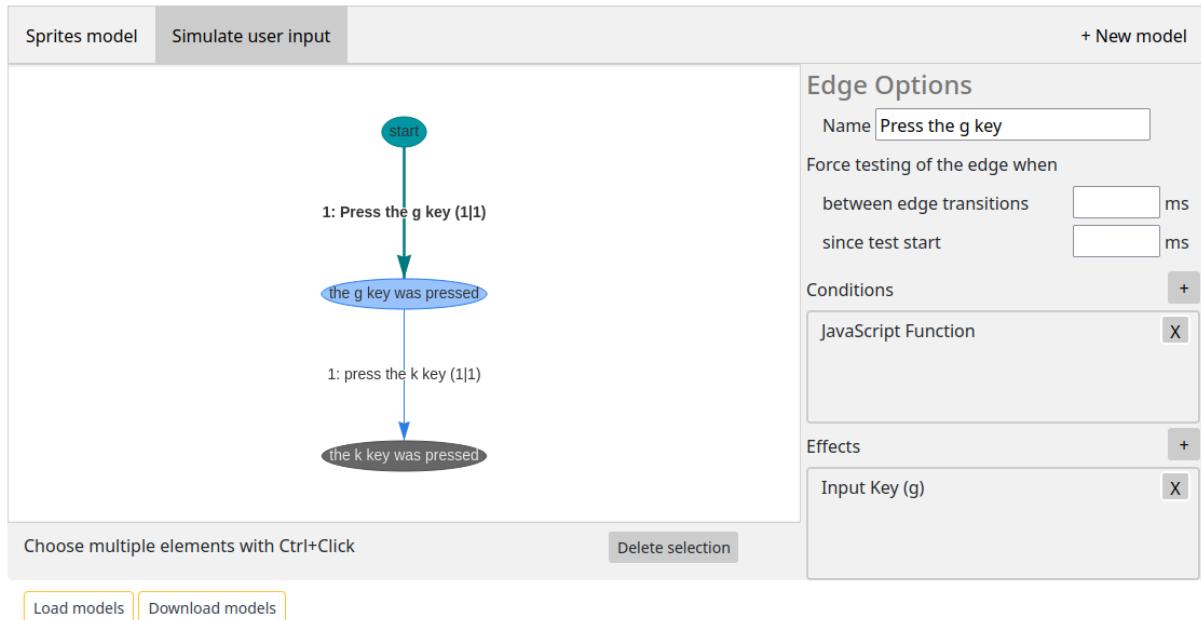
Whisker laat ook toe om testen te schrijven voor Scratchprojecten door het schrijven van eindigetoestandsautomaten (FSA). Figuur 6 en 7 tonen hoe dit er kan uitzien.

- Figuur 6 toont een FSA die interactie van een gebruiker met het Scratchproject simuleert. Eerst wordt de g-toets ingedrukt, dan de k-toets.
- Figuur 7 toont een FSA die het gedrag van de sprites controleert. We zien een simpele FSA met 1 top en 3 gerichte bogen. Door boog 3 te selecteren kunnen we rechts bij “Conditions” zien wanneer deze boog wordt genomen: bij het indrukken van de g-toets. Bij “Effects” zien we het verwachte effect als deze boog genomen wordt: een “Attribute Change” die controleert dat alle sprites (aangeduid door de regex “*”) een vergroting (aangeduid door het “+” symbool) van het attribuut “size” hebben.

De eindigetoestandsautomaten uit Figuur 6 en 7 zijn handmatig gemaakt en worden niet automatisch gegenereerd. Het maken van deze testen is redelijk gebruiksvriendelijk en kan volledig in Whisker. De nadelen zijn dat er al snel geavanceerdere concepten zoals reguliere expressies nodig zijn.

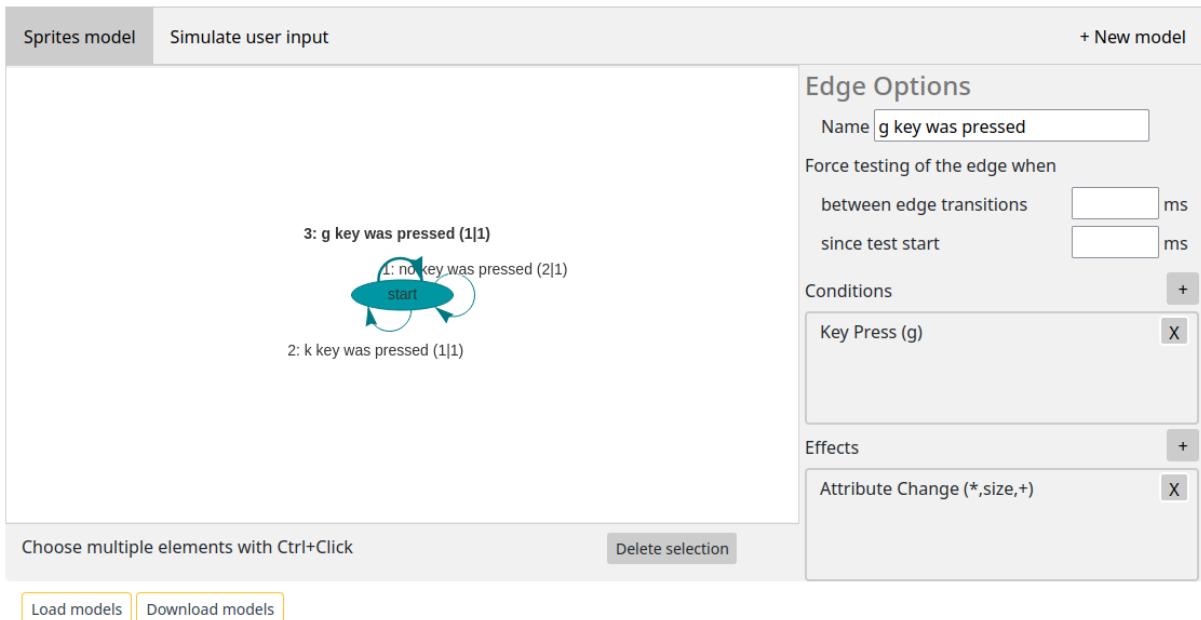
Whisker heeft een aantal mogelijkheden om het testen van Scratchprojecten te faciliteren maar testen schrijven in Scratch zelf is niet mogelijk. Whisker heeft een hogere instapdrempel voor het schrijven van testen dan Poke.

Model Editor



Figuur 6: Een eindigetoestandsautomaat die interactie van een gebruiker met het Scratchproject simuleert.

Model Editor

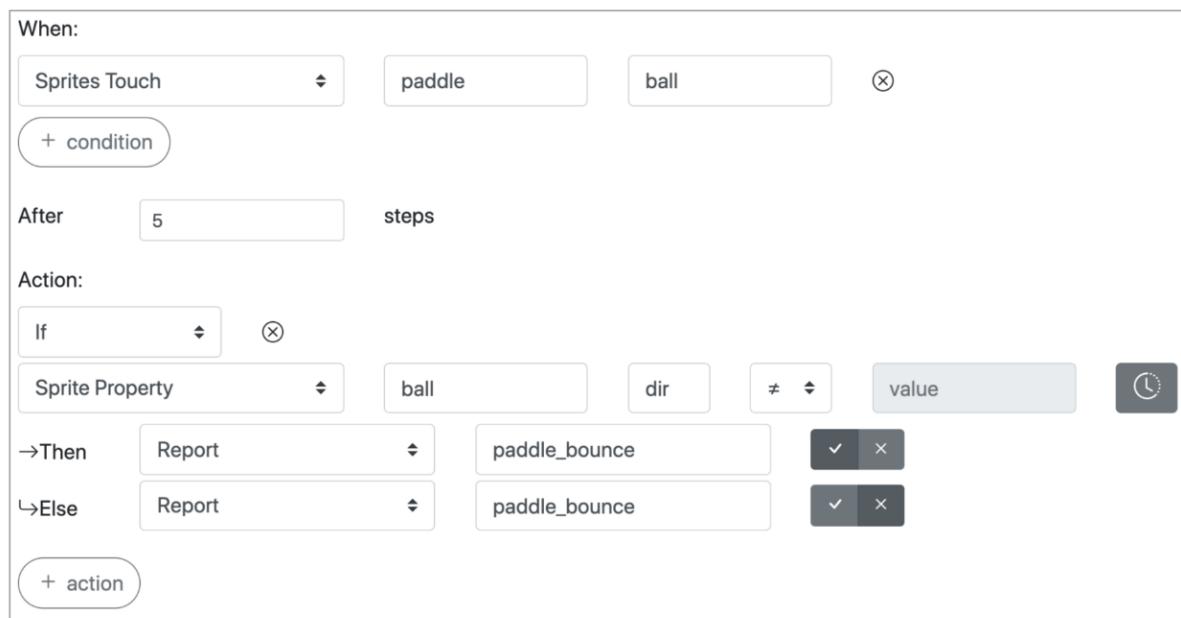


Figuur 7: Een eindigetoestandsautomaat die het gedrag van de sprites controleert.

1.3.4 SnapCheck

SnapCheck (Wang e.a. 2021b) is een testframework voor de blokgebaseerde programmeertaal Snap! (Harvey e.a. 2013). SnapChecktesten worden geschreven in JavaScript of visueel met een interactieve gebruikersomgeving.

In de paper “Snapcheck: Automated Testing for Snap! Programs” (Wang e.a. 2021a) wordt een Snap!-programma getest dat een versie van het spel “Pong”¹ implementeert. Daar wordt een afbeelding getoond van de gebruikersomgeving bij het opstellen van de feedback “paddle_bounce”. Deze afbeelding is te zien in Figuur 8.



Figuur 8: Een voorbeeld van de Snap! gebruikersomgeving waar .

In Figuur 8 wordt er gewacht tot de bal de peddel aanraakt. Daarna wordt er 5 stappen gewacht. En uiteindelijk wordt er positieve feedback gegeven als de richting van de bal veranderde, en negatieve feedback als de richting hetzelfde bleef.

SnapCheck laat toe om testen te schrijven in een interactieve gebruikersomgeving wat de instapdrempel voor leerkrachten kan verlagen. SnapCheck laat niet toe om testen te schrijven in Snap! zelf.

¹<https://ponggame.org>

1.4 Conclusie

Het testen van blokgebaseerde programmeertalen kan op veel verschillende manieren gebeuren. Dat gezegd zijnde bestaat er nog geen testframework waarmee een blokgebaseerde programmeertaal kan getest worden in de blokgebaseerde taal zelf, waar dit voor tekstuele programmeertalen wel de gebruikelijke manier van werken is.

Deze masterproef onderzoekt of het mogelijk is om een testframework te maken waarmee Scratch-projecten in Scratch zelf kunnen getest worden. Ook willen we onderzoeken welke blokken moeten toegevoegd worden om het schrijven van testen zo ergonomisch mogelijk te maken. We streven niet naar een volledig testframework die alles kan testen.

We starten met een overzicht van alle blokken die de Poke-uitbreiding toevoegt in Hoofdstuk 2. In Hoofdstuk 3 wordt besproken hoe een uitbreiding toegevoegd wordt aan de Scratchomgeving door de Poke-uitbreiding als voorbeeld te bekijken. Hoofdstuk 4 gaat technisch in op Poke. Daar bekijken we hoe de functionaliteit die de Pokeblokken bieden geïmplementeerd werd. Daarna, in Hoofdstuk 5, worden voorbeelden gegeven van Scratchprojecten die getest worden met Poke. Uiteindelijk bespreken we toekomstig werk en geven we een conclusie.

2 Poke

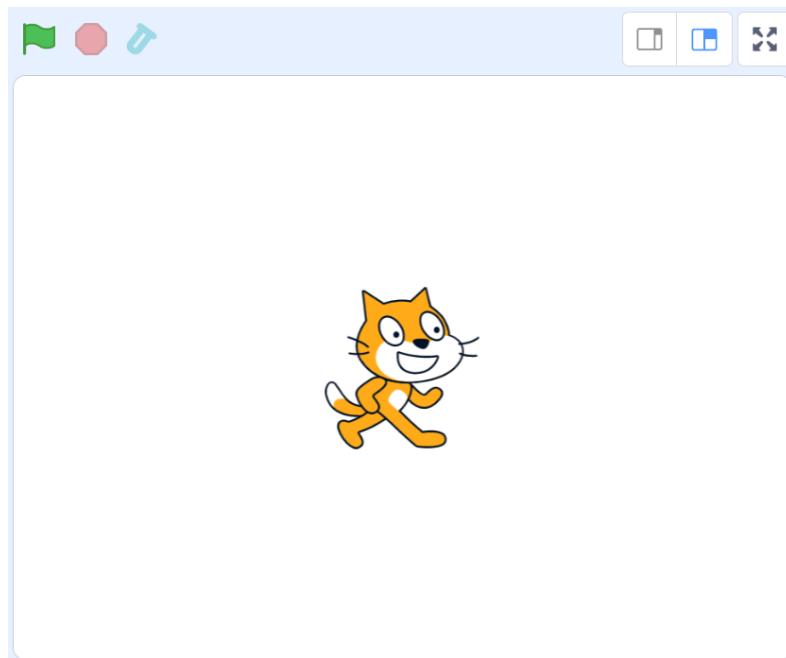
Poke is een uitbreiding voor Scratch die nieuwe blokken toevoegt waarmee testcode kan geschreven worden om de programmacode van een Scratchproject te beoordelen. Op de UGent Scratchomgeving² kan de Poke-uitbreiding toegevoegd worden door linksonder op de knop “Voeg een uitbreiding toe” (Figuur 9) te drukken en Poke te selecteren.



Figuur 9: De knop die het “uitbreidingen” menu opent.

De Poke-uitbreiding voegt ook een blauwe proefbus toe in de Scratch GUI (Figuur 10). Deze knop dient om de testcode uit te voeren. Als op de knop gedrukt wordt, worden alle *when tests started* blokken geactiveerd.

De groene vlag en de rode “stop”-knop zijn bestaande knoppen. De groene vlag activeert alle *when green flag pressed* blokken, de rode “stop”-knop stopt de programmacode en de testcode.



Figuur 10: Het Scratch canvas met de knoppen die er boven staan.

Het *when tests started* blok (Figuur 11) is een startblok. Het is gelijkaardig aan het *when green flag clicked* blok dat al bestaat in Scratch. Een testplan moet altijd beginnen met dit blok.

²<https://scratch.ugent.be/poke>



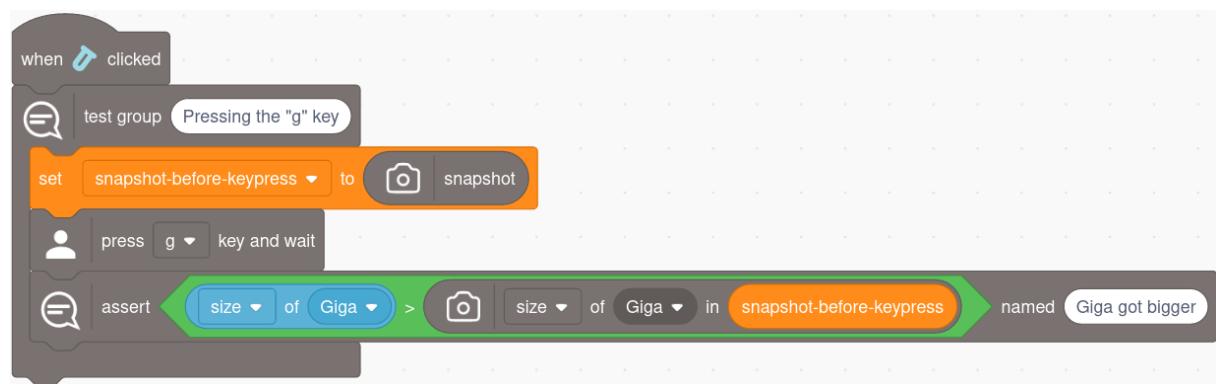
Figuur 11: Het *when tests started* blok

De andere blokken die Poke toevoegt, kunnen elk onderverdeeld worden in één van vier categorieën:

- Feedback: genereren feedback tijdens het uitvoeren van de testen
- Gebruikersinteractie: simuleren interactie van een gebruiker tijdens het uitvoeren van de programmacode van het Scratchproject
- Waarneming: staan ons toe om waarnemingen te doen over de Scratchomgeving tijdens het uitvoeren van de programmacode van het Scratchproject
- Uitvoeren van Blokken in een Andere Sprite (UBAS): voeren blokken uit in andere sprites

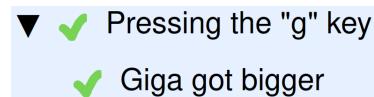
2.1 Een eenvoudig voorbeeld

Figuur 12 toont een voorbeeld van een testplan dat gebruikmaakt van de Poke-uitbreiding. Dit testplan test een deel van de oefening “groeien en krimpen” (uitleg in Paragraaf 1.2). De testcode controleert dat de sprite “Giga” groter wordt als op de g-toets gedrukt wordt. In Figuur 13 is de feedbackboom te zien die deze test genereert bij het controleren van de modeloplossing. Deze feedbackboom heeft 1 wortel en 1 blad. Figuur 14 toont de feedback als de sprite “Giga” niet groter wordt bij het indrukken van de g-toets.

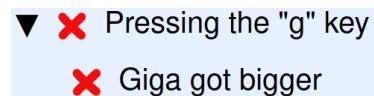


Figuur 12: Deze test controleert of sprite “Giga” groter wordt na het indrukken van de g-toets.

Het testplan begint met een startblok dat geactiveerd wordt als de testcode uitgevoerd wordt. Door het testplan en de feedback te vergelijken, zien we dat het *test group* blok een testgroep aanmaakt in



Figuur 13: De feedback die de testcode in Figuur 13 genereert bij het controleren van een modeloplossing voor de oefening “groei en krimpen”.



Figuur 14: De feedback die de testcode in Figuur 12 genereert bij de oefening “groei en krimpen” na het lopen van de testcode als de sprite “Giga” niet groter wordt als de g-toets ingedrukt wordt.

de feedbackboom. Daarna zien we dat het *snapshot* blok gebruikt wordt om een snapshot van de omgeving op te slaan in een variabele *snapshot-before-keypress*. Verder zien we dat de g-toets indrukken gesimuleerd wordt via het *press key and wait* blok. We komen in Paragraaf 2.3.2 terug op de specifieke betekenis van *and wait* in de naam van dit blok. Uiteindelijk wordt een assert gebruikt om binnen de testgroep feedback te geven over een voorwaarde. De voorwaarde controleert dat de sprite “Giga” groter is geworden na het indrukken van de g-toets door het opgeslagen snapshot te bevragen.

2.2 Feedbackblokken

De feedbackblokken maken het mogelijk om feedback te geven tijdens het uitvoeren van een testplan. Deze blokken zullen tijdens het uitvoeren van de testcode de feedback opbouwen zodat deze na het uitvoeren van de testen kan getoond worden.

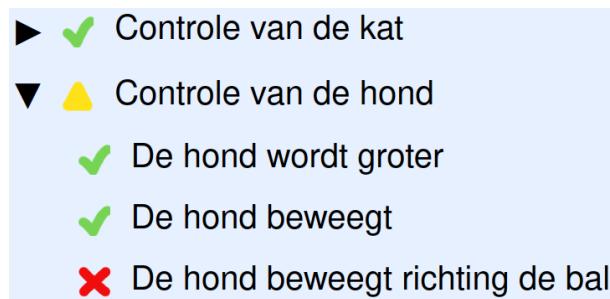
Figuur 13 toonde al een weergave van een eenvoudige feedbackboom. Figuur 15 toont een iets ingewikkeldere feedbackboom. We zien op Figuur 15 ook de 3 verschillende iconen:

- Een groen vinkje: alles verliep goed.
- Een geel driehoekje: niet alle feedback in de testgroep heeft een groen vinkje.
- Een rood kruisje: er ging iets mis.

2.2.1 Een blad toevoegen aan de feedbackboom

Het *assert* blok (Figuur 16) controleert een voorwaarde en geeft feedback die aangeeft of de voorwaarde voldaan is. Dit blok neemt 2 argumenten:

- Een Booleaanse voorwaarde: als deze waar is zal er een groen vinkje in de feedback komen, anders wordt dit een rood kruisje.



Figuur 15: Een voorbeeld feedbackboom. De volledig correcte testgroep is automatisch dichtgeklapt. De testgroep waar iets mis ging krijgt een geel driehoekje.

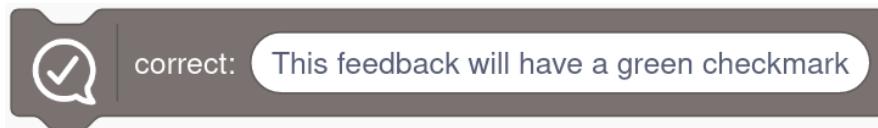
- De tekst die zal toegevoegd worden aan de feedback: een heldere beschrijving van de Booleaanse conditie zodat het duidelijk is voor een leerling wat er gecontroleerd werd.

In Figuur 16 zal het *assert* blok de tekst “42 is equal to 42” toevoegen aan de feedbackboom met een groen vinkje omdat de voorwaarde correct is.



Figuur 16: Het *assert* block.

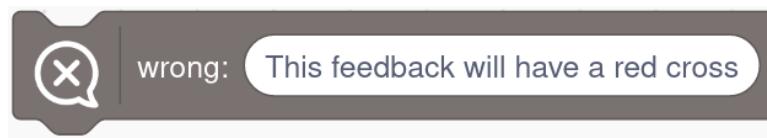
Het *correct* blok (Figuur 17) en *wrong* blok (Figuur 18) voegen respectievelijk correcte en foute feedback toe aan de feedbackboom. Het *correct* blok zal feedback met een groen vinkje geven terwijl het *wrong* blok feedback met een rood kruisje geeft. Figuur 19 toont Scratchcode die dezelfde feedback als het *assert* blok in Figuur 16 zal genereren. Het *assert* blok is dus eigenlijk een vereenvoudiging van deze constructie die geen aparte feedback gebaseerd op het resultaat van de voorwaarde toelaat.



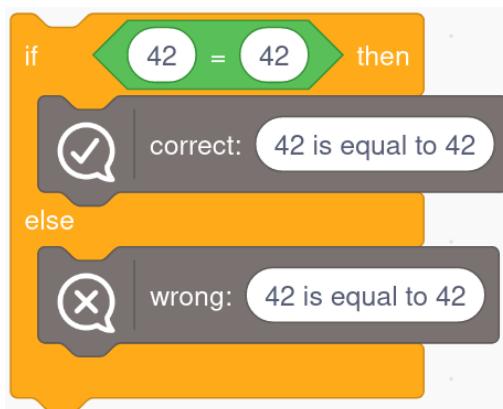
Figuur 17: Het *correct* block.

2.2.2 Interne toppen toevoegen aan de feedbackboom

Het *test group* blok (Figuur 20) is een C-blok. Dit blok laat toe om de feedback een boomstructuur te geven. Het groepeert een stuk testcode en voegt alle feedback van deze testcode toe aan een groep. In

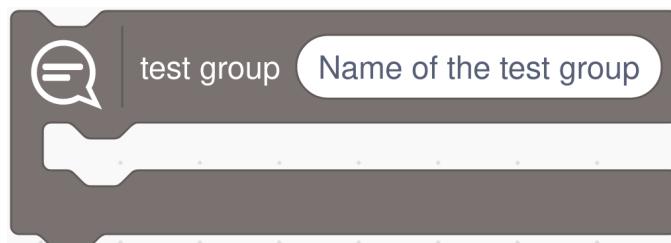


Figuur 18: Het *wrong* block.



Figuur 19: Scratchcode dat gebruik maakt van het *correct* blok en *wrong* blok om het gedrag van het *assert* blok in Figuur 16 na te bootsen.

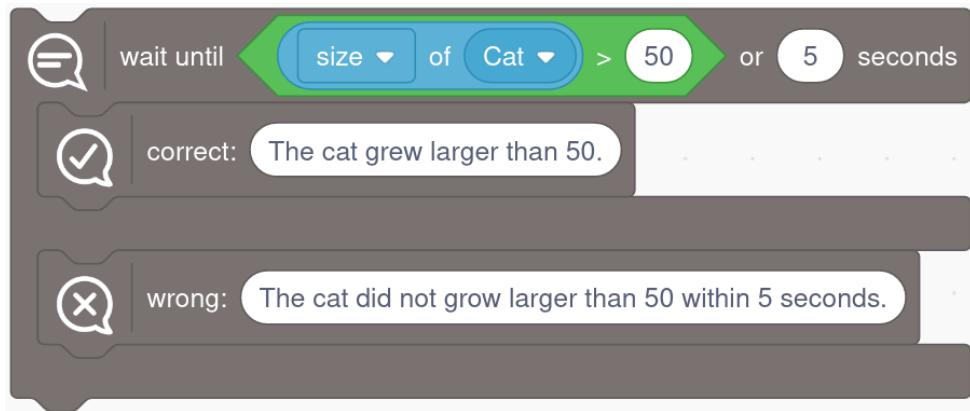
de weergave van de feedbackboom kunnen takken open- en dichtklappen. Dit blok heeft 1 argument: de naam van de groep.



Figuur 20: Het *test group* blok

2.2.3 Wachten tot iets gebeurt

Het *wait until or timeout* blok (Figuur 21) blok is een C-blok met 2 takken. Dit blok zal wachten tot de conditie voldaan is, maar verwijdt dat er oneindig lang gewacht wordt. In het voorbeeld van Figuur 21 zal er gewacht worden tot de grootte van de sprite “Cat” groter dan 50 is. Als het wachten langer dan 5 seconden duurt stopt het wachten en wordt de tweede tak uitgevoerd, hier het *wrong* blok. Als er zich geen time-out voordoet wordt de eerste tak uitgevoerd.



Figuur 21: Het *wait until or timeout* blok

De time-out functionaliteit staat ons toe om te kunnen wachten op een gebeurtenis zonder dat de testcode oneindig lang wacht als de gebeurtenis zich niet voordoet.

2.3 Gebruikersinteractieblokken

Gebruikersinteractieblokken dienen om tijdens het uitvoeren van de testcode te simuleren hoe, waar en wanneer een gebruiker met Scratch interageert. De gebruikersinteractieblokken bevatten blokken om de muis en het toetsenbord te simuleren.

2.3.1 Threads

Alle Scratchcode die uitgevoerd wordt is deel van een thread. Een thread wordt gecreëerd als gevolg van gebruikersinteractie die opgevangen wordt door een startblok. Bijvoorbeeld: als er op de groene vlag wordt gedrukt, wordt voor elke stapel blokken die een *when green flag pressed* startblok heeft een thread aangemaakt waar de blokken van de stapel in uitgevoerd zullen worden.

2.3.2 *and wait* blokken

Blokken die interactie simuleren starten vaak startblokken in de programmacode. In die gevallen moet er tijdens het uitvoeren van de testcode gewacht worden tot de alle gestarte threads in de programmacode uitgevoerd werden.

In Paragraaf 2.1, Figuur 12 wordt het indrukken van de g-toets gesimuleerd voor de oefening “groei en krimpen”. Bij de modeloplossing zorgt dit ervoor dat de sprite “Giga” groter wordt. Het *assert* blok mag pas controleren of deze vergroting gelukt is als de programmacode van “groei en krimpen”

(zie Figuur 3) de grootte van de sprite heeft kunnen veranderen. Als er niet gewacht wordt, dan zou het kunnen zijn dat het *assert* blok uitgevoerd wordt net voor deze verandering. Dit leidt tot een verkeerde waarneming waardoor foute feedback wordt gegeven.

Net zoals in Scratch de blokken *broadcast* en *broadcast and wait* bestaan voegt Poke de volgende paren van blokken toe:

- *press key* en *press key and wait*
- *click sprite* en *click sprite and wait*

Deze paren hebben telkens een blok dat interactie simuleert zonder te wachten in de testcode en een overeenkomstig blok dat deze interactie doet en daarna wel wacht.

2.3.3 Press green flag

Het *press green flag* blok (zie Figuur 22) simuleert dat er op de groene vlag gedrukt wordt.



Figuur 22: Het *press green flag* blok

2.3.4 Press key

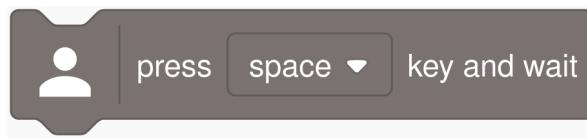
Het *press key* blok (zie Figuur 23) simuleert dat er op een toets gedrukt wordt. Tijdens de uitvoering simuleert dit blok een toetsaanslag en wordt daarna verdergegaan naar de uitvoering van het volgende blok in de testcode.



Figuur 23: Het *press key* blok

2.3.5 Press key and wait

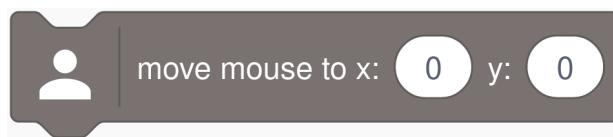
Het *press key and wait* blok (zie Figuur 24) simuleert dat er op een toets gedrukt wordt. In tegenstelling tot het *press key* blok wacht dit blok tot alle threads die gestart zijn in de programmacode door de toetsaanslag klaar zijn met uitvoeren.



Figuur 24: Het *press key and wait* blok

2.3.6 Move mouse to

Het *move mouse to* blok (zie Figuur 25) kan een muis beweging simuleren. Het blok neemt een x- en y-waarde van de coördinaat (x,y) op het canvas naar waar de muisaanwijzer moet verplaatst worden.



Figuur 25: Het *move mouse to* blok

2.3.7 Click sprite

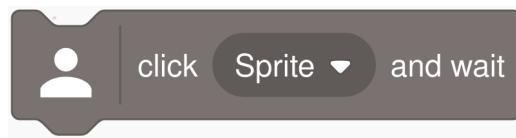
Het *click sprite* blok (zie Figuur 26) simuleert dat er op een sprite geklikt wordt. Tijdens de uitvoering simuleert dit blok een muisklik op een sprite. Na het klikken wordt er direct verder gegaan naar de uitvoering van het volgende blok in de testcode.



Figuur 26: Het *click sprite* blok

2.3.8 Click sprite and wait

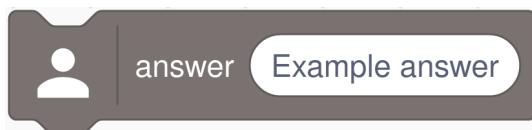
Het *click sprite and wait* blok (zie Figuur 27) simuleert dat er op een sprite geklikt wordt. In tegenstelling tot het *click sprite* blok wacht dit blok tot alle threads die gestart zijn in de programmacode door de toetsaanslag klaar zijn met uitvoeren.



Figuur 27: Het *click sprite and wait* blok

2.3.9 Answer

Het *wait for question and answer* blok (zie Figuur 28) kan een antwoord geven op een vraag. Als er geen vraag gesteld wordt op het moment dat dit blok wordt uitgevoerd zal de testcode wachten tot er een vraag gesteld wordt.



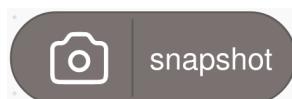
Figuur 28: Het *answer* blok

2.4 Waarnemingsblokken

Waarnemingsblokken dienen om waarnemingen te doen over de omgeving op een bepaald punt in de tijd.

2.4.1 Snapshots nemen

Het *snapshot* blok (zie Figuur 29) is een verslaggeversblok. De waarde die dit blok teruggeeft, is een snapshot van de omgeving. Dit blok kan gebruikt worden om snapshots op te slaan in Scratchvariabelen en daar later informatie uit te halen.



Figuur 29: Het *snapshot* blok

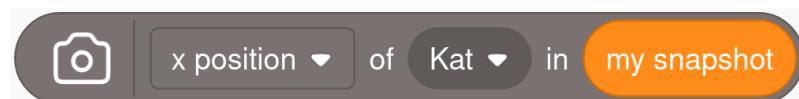
2.4.2 Informatie uit snapshots halen

Het *query snapshot* blok (zie Figuur 30) is een verslaggeversblok dat informatie uit een snapshot kan halen. Het heeft 3 argumenten:

- Een eigenschap van een sprite
- Een sprite van een snapshot
- Een snapshot

De eigenschappen die per sprite kunnen opgevraagd worden zijn:

- x position: de x positie
- y position: de y positie
- direction: de richting
- costume #: het nummer van het uiterlijk
- costume name: de naam van het uiterlijk
- size: de grootte
- volume: het volume
- saying: wat de sprite aan het zeggen was (lege string als dit niets was)
- thinking: wat de sprite aan het denken was (lege string als dit niets was)



Figuur 30: Het query snapshot blok.

In Figuur 30 wordt de x-positie van de sprite “Kat” opgevraagd uit de snapshot dat opgeslagen zit in de Scratchvariabele “my snapshot”.

2.5 Uitvoeren van Blokken in een Andere Sprite (UBAS)

Testcode wordt in één sprite geschreven om een duidelijker overzicht te hebben. Maar het is het vaak nuttig om code te laten uitvoeren door een andere sprite. Bij het testen van de oefening “groeien en krimpen” kan het bijvoorbeeld zijn dat de testcode de grootte van de sprites wil resetten. In Scratch is het niet mogelijk om de grootte van een andere sprite aan te passen, dus dit kan gedaan worden met UBAS.

We zeggen er hier direct bij dat Scratchblokken uitvoeren in naam van een andere sprite ingaat tegen het Scratchmodel van sprites met hun eigen Scratchcode, maar dit geeft wel veel mogelijkheden tijdens het testen. Het alternatief is dat we de testcode verspreiden over verschillende sprites en deze aansturen met signalen. Dit zorgt voor veel omslachtigere testcode. Figuur 31 toont het gebruik van een signaal om de sprite “Kat” van positie te veranderen. Deze Scratchcode moet in de sprite “Kat” staan. Een andere sprite (waar de testcode staat) kan dan het signaal “verplaats Kat” sturen om de kat op positie (0, 0) te zetten.

Nu bekijken we een betere manier om dit te doen met een UBAS-blok van de Poke-uitbreiding.



Figuur 31: Een signaal gebruiken om de positie van de sprite “Kat” te veranderen. Deze Scratchcode bevindt zich in de sprite “Kat” zelf.

2.5.1 With sprite do

Het *with sprite do* blok (zie Figuur 32) is een C-blok. De blokken in dit C-blok worden door een andere sprite uitgevoerd. Merk op dat de selectie van de sprite gebeurt door een vervangbaar menu. Dit betekent dat hier ook variabelen kunnen gebruikt worden om tijdens de uitvoering de sprite te bepalen.



Figuur 32: Het *with sprite do* blok

In Figuur 32 zal de sprite “Kat” op positie (0, 0) gezet worden.

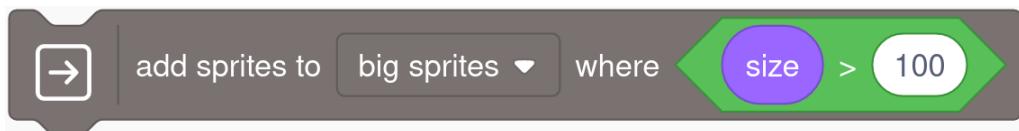
2.5.2 Sprites selecteren met een filter

In Paragraaf 2.5.1 haalden we aan dat sprite injectie kon gedaan worden bij sprites die tijdens uitvoering bepaald werden. Het *sprite filter* blok (zie Figuur 33) kan daarbij helpen. Dit blok laat toe een subset te maken van de sprites en deze in een lijst op te slaan. Deze subset wordt gemaakt door de sprites te filteren gebaseerd op een Booleaanse conditie.

Het *sprite filter* blok heeft 2 argumenten:

- Een lijst waar de resulterende subset van sprites in wordt opgeslagen.
- Een voorwaarde waaraan de geselecteerde sprites moeten voldoen. Deze voorwaarde zal uitgevoerd worden in elke sprite, en als deze naar **true** evalueert wordt de sprite toegevoegd aan de lijst. In Figuur 33 betekend dit dat het *size* blok dus telkens de grootte van de sprite waar

de voorwaarde in gecontroleerd wordt zal teruggeven. En niet de grootte van de sprite waar de testcode in staat.



Figuur 33: Het *sprite filter* blok

In Figuur 33 zullen alle sprites die groter zijn dan 100 terecht komen in de lijst `big sprites`.

2.6 Tips bij het schrijven van testen

Hier worden een aantal handige tips gegeven die kunnen gebruikt worden tijdens het schrijven van Poke testen.

2.6.1 Testcode in een aparte sprite

Het is aangeraden een aparte sprite te voorzien voor de testcode. Dit voorkomt dat er testcode geschreven wordt die alleen werkt als ze uitgevoerd wordt in 1 sprite. Ook zorgt dit voor een duidelijke scheiding tussen test- en programmacode.

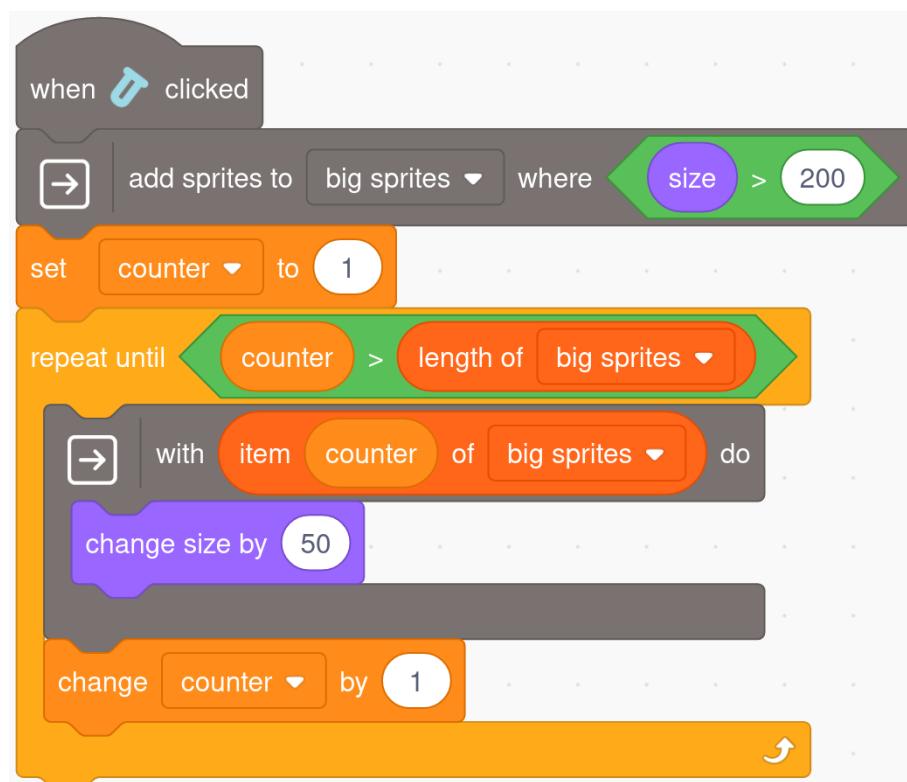
Testcode in een aparte sprite houden heeft ook als voordeel dat variabelen gebruikt door de testcode sprite specifiek kunnen zijn. Als een lus in de testcode een variabele “teller” nodig heeft, dan kan deze variabele zodanig ingesteld worden dat deze alleen beschikbaar is in de testsprite. Als de programmacode ook een variabele “teller” gebruikt zal de testcode niet voor interferentie zorgen.

2.6.2 Dezelfde code uitvoeren in meerdere sprites

Een combinatie van het *sprite filter* blok en het *with sprite do* blok kan gebruikt worden om sprites te selecteren waar dezelfde code moet in uitgevoerd worden. Figuur 34 toont een voorbeeld waar alle sprites die groter zijn dan 200 geselecteerd worden en in de lijst “big sprites” terecht komen. Daarna overlopen we de lijst en wordt met één *with sprite do* blok elke sprite vergroot.

2.6.3 Interferentie vermijden

Vermijd interferentie met het uitvoeren van de programmacode. Testcode die niet verantwoordelijk is voor de setup of teardown van de omgeving zou geen rechtstreekse veranderingen mogen aanbrengen op de omgeving. Enkele voorbeelden van rechtstreekse veranderingen van de omgeving zijn:

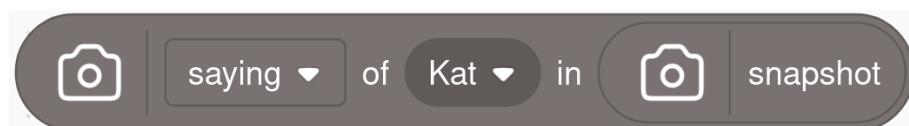


Figuur 34: Een combinatie van het *sprite filter* blok en het *with sprite do* blok.

- Het verplaatsen van een sprite
- Het veranderen van een uiterlijk
- De waarde van een variabele veranderen die gebruikt wordt door de programmacode

2.6.4 Het *query snapshot* blok in combinatie met het *snapshot* blok gebruiken

Een combinatie van het *snapshot* blok en *query snapshot* blok kunnen gebruikt worden om real-time informatie van de omgeving op te vragen. Figuur 35 toont een voorbeeld van hoe kan opgevraagd worden wat de sprite “Kat” momenteel zegt.



Figuur 35: Een combinatie van het *query snapshot* blok en het *snapshot* blok.

2.7 Ontbrekende functionaliteit

Poke ontbreekt nog enkele functionaliteiten. De uitbreiding is in het Engels geschreven en een vertaling naar Nederlands bestaat nog niet. Ook kan Poke nog niet omgaan met klonen.

2.7.1 Meerdere *when tests started* blokken

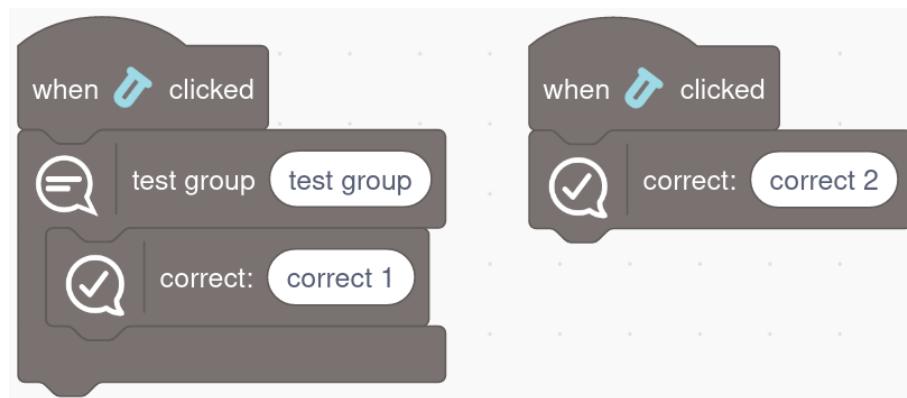
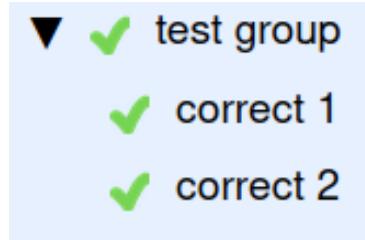
Meerdere *when tests started* blokken kunnen bestaan in een Scratchproject, maar momenteel zal dit tot ongedefinieerd gedrag leiden omdat de threads die ontstaan uit deze *when tests started* blokken parallel zullen uitvoeren. Het parallel uitvoeren van testblokken is niet ondersteund. Dit komt omdat Poke een feedbackboom opstelt tijdens het uitvoeren van de testen. Als testblokken parallel worden uitgevoerd, zal feedback van uit meerdere threads toegevoegd worden aan dezelfde boom.

Figuur 36 toont een voorbeeld van testblokken die parallel uitgevoerd worden.

Figuur 37 toont de feedbackboom dat gegenereerd wordt. We zien dat “correct 2” in de testgroep komt terwijl dit volgens de code niet zou mogen gebeuren.

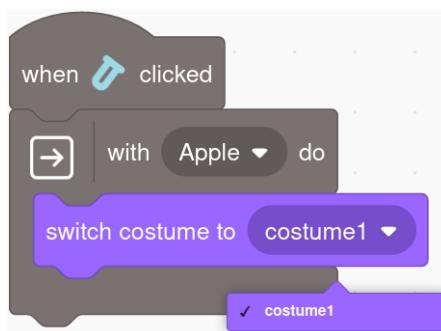
2.7.2 Statisch testen van Scratchblokken

Poke kan niet controleren welke Scratchblokken gebruikt worden in de programmacode. Poke is bedoeld om gedragstesten te schrijven. Gedragstesten controleren het gedrag van een Scratchproject.

**Figuur 36:** Parallelle testcode**Figuur 37:** De feedback als resultaat van het uitvoeren van de testcode te zien in Figuur 36

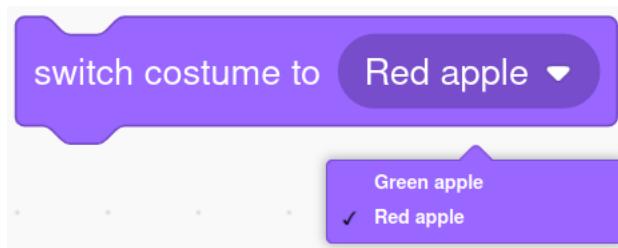
2.7.3 Spritespecifieke editoropties bij UBAS

In Figuur 38 zien we de testcode die zich in de testsprite bevindt. Het menu van het *switch costume to* blok toont de uiterlijken die bij de testsprite horen, hier is er maar 1 uiterlijk, namelijk "costume1".

**Figuur 38:** Testcode die zich in de testsprite bevindt.

In Figuur 39 zien we hetzelfde blok, maar deze staat nu in de "Apple" sprite. In het menu zijn nu 2 uiterlijken te zien: "Green apple" en "Red apple".

Idealiter zouden de menu's te zien in Figuur 38 en Figuur 39 dezelfde moeten zijn, want de blokken

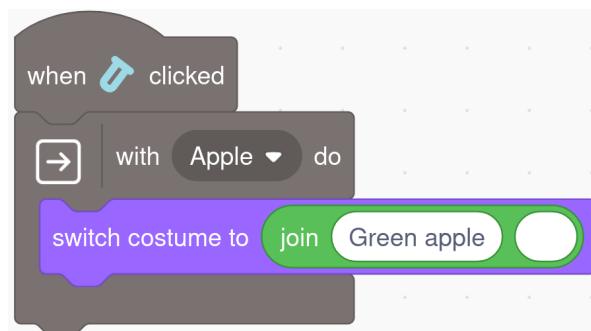


Figuur 39: Testcode die zich in de “Apple” sprite bevindt.

worden in dezelfde sprite uitgevoerd. Maar dit gebeurt nog niet.

Er kan rond deze tekortkoming gewerkt worden door in de testsprite extra uiterlijken toe te voegen. De testsprite zou hier dan bijvoorbeeld 2 extra uiterlijken krijgen met als naam: “Green apple” en “Red apple”. Dit zou ervoor zorgen dat ook deze opties beschikbaar zijn in het menu van het *switch costume to* blok in de testsprite.

Een andere manier om rond dit probleem te werken is te zien in Figuur 40. Het menu wordt vervangen door een verslaggeversblok dat de naam van een uiterlijk doorgeeft.



Figuur 40: Het *join* verslaggeversblok wordt gebruikt om een uiterlijk te selecteren.

2.8 Conclusie

Poke biedt 4 soorten blokken aan. De feedbackblokken worden gebruikt om tijdens het uitvoeren van de testcode de feedbackboom op te stellen. De gebruikersinteractieblokken kunnen interactie van een gebruiker simuleren. Waarnemingsblokken staan de lesgever toe om waarnemingen te doen over de Scratchomgeving. En met UBAS-blokken kan er Scratchcode in naam van een andere sprite uitgevoerd worden.

3 Een uitbreiding toevoegen aan Scratch

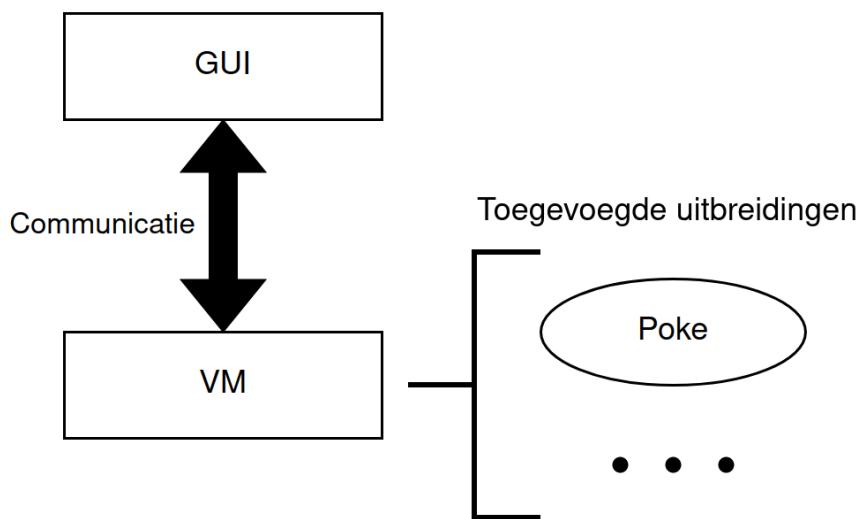
Aan de Scratchomgeving kunnen uitbreidingen toegevoegd worden. Bij het inladen van een uitbreiding wordt een extra blokkencategorie toegevoegd aan de Scratchomgeving. Deze categorie bevat alle blokken die bij de uitbreiding horen.

Er moet voor elk toegevoegd blok een JSON-definitie gemaakt worden die het uitzicht van het blok beschrijft. Het gedrag van een blok wordt geïmplementeerd in een bijhorende methode.

In dit hoofdstuk gebruiken we stukken code uit de Poke-uitbreiding om het toevoegen van een uitbreiding aan Scratch duidelijker uit te leggen. Functionaliteiten die niet gebruikt werden in de Poke-uitbreiding worden hier niet uitgelegd. We leggen ook zaken uit die niet in de officiële documentatie³ te vinden zijn.

3.1 Architectuur

Figuur 41 toont een overzicht van de Scratch GUI, de Scratch VM en de uitbreidingen.



Figuur 41: Een overzicht van de Scratch GUI, de Scratch VM en de uitbreidingen.

Een uitbreiding wordt toegevoegd aan de Scratch VM. De Scratch VM is een deel van de Scratch codebase. Deze is onderverdeeld in twee Githuborganisaties: Scratch⁴ en Scratch Foundation⁵. De twee belangrijke Githubrepositories voor deze masterproef zijn:

³<https://github.com/scratchfoundation/scratch-vm/blob/develop/docs/extensions.md>

⁴<https://github.com/LLK/>

⁵<https://github.com/scratchfoundation/>

- De Scratch GUI: de IDE die kinderen gebruiken om code te schrijven en uit te voeren.
- De Scratch VM: wordt gebruikt door de GUI om de Scratchblokken uit te voeren.

Er is geen platform waarmee derden uitbreidingen kunnen aanbieden voor de Scratchcommunity. Dit betekent dat het toevoegen van een uitbreiding rechtstreeks in de codebase van de Scratch VM moet. Uitbreidingen breiden alleen de Scratch VM uit en bieden geen mogelijkheid om extra lay-outcomponenten aan de Scratchomgeving toe te voegen. Er kan bijvoorbeeld geen extra knop naast de groene vlag toegevoegd worden, dit is een verandering die rechtstreeks in de GUI-code moet gebeuren.

Dit is de bestandsstructuur van de Scratch VM:

```
src/
└── ...
   └── extensions/
      ├── ...
      ├── scratch3_pen/
      ├── scratch3_debugger/
      ├── scratch3_poke/
         ├── index.js
         └── ...
      ...
...
```

Hier zien we dat voor de Poke-uitbreiding een map `src/extensions/scratch3_poke` bestaat. In deze map bevindt zich een bestand `index.js` dat een klasse definieert die bij de uitbreiding hoort.

3.2 Uitbreidingsklasse

Een uitbreidingsklasse definieert blokken die samen een uitbreiding vormen en implementeert hun functionaliteit. Ook kunnen nieuwe dropdown menu's gedefinieerd en geïmplementeerd worden.

De uitbreidingsklasse van een uitbreiding heeft de volgende methoden:

- `getInfo`: geeft een configuratie (JSON) terug waar onder andere het uiterlijk van de toegevoegde blokken gedefinieerd wordt.
- `implementatieBlok1`: implementeert de functionaliteit van het eerste blok.
- ...

- `implementatieBlokN`: implementeert de functionaliteit van het N-de blok.
- `implementatieMenu1`: geeft een lijst terug die de waarden van het eerste menu bevat. Nde
- ...
- `implementatieMenuN`: geeft een lijst terug die de waarden van het N-de menu bevat.

De constructor van de klasse krijgt het runtime-object van de Scratch VM mee. Het runtime-object bevat alle informatie over de Scratch VM. Enkele voorbeelden zijn:

- Een lijst met alle lopende threads
- De sprites en hun eigenschappen

Een voorbeeld van de uitbreidingsklasse is te zien in de volgende code:

```
1 class Scratch3PokeBlocks {
2     constructor (runtime) {
3         this.runtime = runtime;
4     }
5
6     getInfo () {
7         return {
8             id: 'poke',
9             color1: '#797270',
10            color2: '#605B59',
11            color3: '#605B59',
12            name: 'Testing',
13            blocks: [
14                // Een JSON-definitie van elk blok.
15            ],
16            menus: {
17                // Een JSON-definitie voor uitbreidingspecifieke drop-
18                // down menu's.
19            }
20        }
21    }
}
```

Merk op dat in de constructor het runtime object wordt opgeslagen voor later gebruik.

De sleutelwaardeparen die in de configuratie (JSON) zitten zijn:

sleutel waarde

id Prefix voor alle opcodes van blokken van de uitbreiding.

color1 De achtergrond kleur van alle blokken in de uitbreiding (in hex waarde).

color2 De kleur van vervangbare menus van alle blokken in de uitbreiding (in hex waarde).

sleutel	waarde
---------	--------

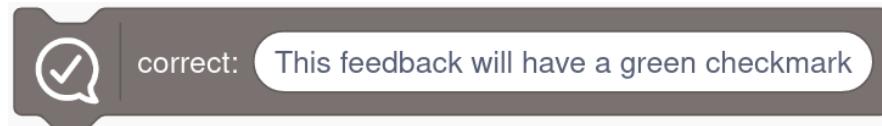
color3	De kleur van de randen van alle blokken in de uitbreiding en de achtergrondkleur van velden waar Booleaanse verslaggeversblokken in passen. (in hex waarde).
name	Naam van de uitbreiding zoals gebruikt in de UI van Scratch. Dit is optioneel en heeft als default waarde het id
blocks	Een lijst van de JSON-definities per blok.
menus	Een object dat verschillende menu's kan definiëren die de uitbreidingsblokken kunnen gebruiken.

3.3 Een blok definiëren

Hieronder is de JSON-definitie van het *correct* blok te zien:

```
1  {
2      opcode: 'addCorrectFeedback',
3      blockIconURI: feedbackCorrectIcon,
4      blockType: BlockType.COMMAND,
5      text: 'correct: [FEEDBACK]',
6      arguments: {
7          FEEDBACK: {
8              type: ArgumentType.STRING,
9              defaultValue: ''
10         }
11     }
12 }
```

Het resulterende *correct* blok:



De volgende 4 sleutelwaardeparen zijn essentieel voor elk blok:

sleutel	waarde
---------	--------

opcode	Het suffix van de opcode van het blok. Het prefix zal de id waarde van de uitbreiding zijn gevolgd door een underscore. Hier zal de opcode van het <i>correct</i> blok dus poke_addCorrectFeedback zijn.
--------	--

sleutel	waarde
blockType	Er zijn verschillende types blokken. Het <i>correct</i> blok is een commandoblok en heeft dus het type COMMAND . De andere types worden later besproken.
text	De tekst- en argumentdefinitie van het blok. Hier wordt bepaald welke tekst op een blok staat en waar de argumenten zich bevinden.
arguments	Object dat de naam van een argument maapt op de definitie van het argument.

Het sleutelwaardepaar **blockIconURI** is optioneel en voegt een icoon toe aan de start van het blok zoals de tekstballon bij het *correct* blok. De waarde is een variabele die als volgt gedefinieerd wordt:

```
1 const feedbackCorrectIcon = require('./images/icon--feedback-correct.svg');
```

Het *correct* blok voegt correcte feedback toe aan de feedbackboom. Dit wordt gedaan in een methode die bij de Scratch3PokeBlocksklasse hoort. Een methode waar de functionaliteit van een blok geprogrammeerd wordt, noemen we een functionaliteitsmethode. Een functionaliteitsmethode moet dezelfde naam hebben als de opcode die meegegeven wordt in de JSON-definitie, in dit voorbeeld dus **addCorrectFeedback**:

```
1 addCorrectFeedback (args) {
2     // code die args.FEEDBACK gebruikt om de feedbackboom aan te vullen
3 }
```

Tekens als het *correct* blok wordt uitgevoerd in de Scratchcode, zal achterliggend de methode **addCorrectFeedback** uitgevoerd worden. De **addCorrectFeedback** krijgt een object **args** mee. **args.FEEDBACK** zal de waarde van de **FEEDBACK** variabele bevatten.

3.3.1 Een startblok

Hieronder is de JSON-definitie van het *when tests started* blok te zien:

```
1 {
2     opcode: 'startTests',
3     blockType: BlockType.HAT,
4     isEdgeActivated: false,
5     shouldRestartExistingThreads: true,
6     text: 'when [IMAGE] clicked',
7     arguments: {
8         IMAGE: {
9             type: ArgumentType.IMAGE,
10            dataURI: testTubeIcon
11        }
12    }
13}
```

```

11         }
12     }
13 }
```

Het resulterende *when tests started* blok:



Hier zien we dat het bloktype nu **HAT** is. Verder is er een sleutelwaardepaar `shouldRestartExistingThreads` toegevoegd. Als de waarde **true** is zullen bestaande threads die aangemaakt werden door dit startblok gestopt worden als dit startblok nogmaals wordt uitgevoerd. Als de waarde **false** is moeten bestaande threads eerst eindigen. Standaard staat dit op **false**.

Ook is er een sleutelwaardepaar `isEdgeActivated` toegevoegd. Figuur \ref{fig:when_greater_than} toont een voorbeeld van een startblok dat "edge activated" is. Het \textit{when greater than} startblok wordt uitgevoerd als de voorwaarde "loudness > 10" voldaan is. De functionaliteitsmethode die bij het \textit{when greater than} blok hoort, controleert de voorwaarde "loudness > 10". Deze methode wordt constant uitgevoerd doorheen de tijd. Een voorbeeld van een verloop van de **return**-waarden doorheen de tijd van deze methode is te zien in Figuur \ref{fig:edgeActivated}. Het startblok activeert hier één keer: als de **return**-waarde van `falsenaartrue` gaat, ofwel als de voorwaarde "loudness > 10" waar wordt.

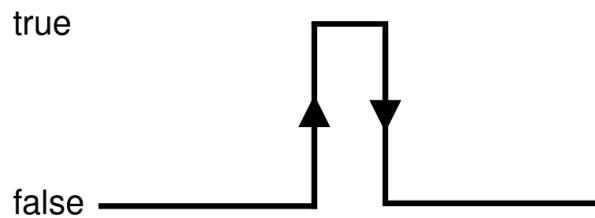


Figuur 42: Het *when greater than* blok.

Het *when tests started* blok moet uitgevoerd worden als op de testknop gedrukt wordt. Op de testknop drukken gooit een Scratchgebeurtenis "PROJECT_TESTS_START" op. In de constructor van de uitbreidingsklasse staat code die ervoor zorgt dat deze gebeurtenis opgevangen wordt, en alle *when tests started* blokken gestart worden:

```

1 class Scratch3PokeBlocks {
2     constructor (runtime) {
3         this.runtime = runtime;
4         this.runtime.on('PROJECT_TESTS_START', () => {
5             this.runtime.startHats('poke_startTests');
```



Figuur 43: De return-waarde van de functionaliteitsmethode van een “edge activated” startblok doorheen de tijd.

```

6
7      });
8  }
9  ...
10 }
```

Het *when tests started* blok moet dus geen constante evaluatie van zijn functionaliteitsmethode doen want het blok zal gestart worden bij het voorkomen van de “PROJECT_TEST_START” Scratchgebeurtenis. Daarom staat `isEdgeActivated` op **false**.

De functionaliteitsmethode van het *when tests started* blok ziet er als volgt uit:

```

1 startTests () {
2     return true;
3 }
```

Als hier geen **true** wordt teruggegeven zal het startblok alsnog niet starten.

3.3.1.1 Iconen

Het *when tests started* blok gebruikt een icoon (de proefbuis). De afbeelding wordt toegevoegd via een argument dat we hier `IMAGE` noemen. De waarde van het sleutelwaardepaar `dataURI` is op dezelfde manier gemaakt als de waarde van het sleutelwaardepaar `blockIconURI` in Paragraaf 3.3.

3.3.2 Een C-blok

Hieronder is de JSON-definitie van het *test group* blok te zien:

```

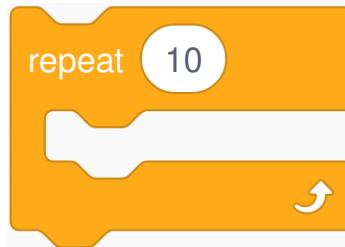
1 {
2     opcode: 'groupName',
3     blockIconURI: feedbackBlockIcon,
4     blockType: BlockType.CONDITIONAL,
5     branchCount: 1,
6     text: 'test group [GROUP_NAME]',
```

```
7     arguments: {
8         GROUP_NAME: {
9             type: ArgumentType.STRING,
10            defaultValue: ''
11        }
12    }
13 }
```

Het resulterende *test group* blok:



Hier zien we dat het bloktype nu **CONDITIONAL** is. Dit zorgt voor de C-vorm van het blok. Als het bloktype **LOOP** is zal het blok ook een C-vorm krijgen en zal een extra icoon toegevoegd worden:



Het sleutelwaardepaar **branchCount** beslist hoeveel takken er zijn. Het *if/else* blok heeft 2 takken:



3.3.3 Verslaggeversblokken

Hieronder is de JSON-definitie van het *snapshot* blok te zien:

```
1 {
2   opcode: 'snapshot',
3   blockIconURI: observeBlockIcon,
4   blockType: BlockType.REPORTER,
5   text: 'snapshot',
6   arguments: {}
7 }
```

Het resulterende *snapshot* blok:



Hier zien we dat het bloktype nu **REPORTER** is omdat het *snapshot* een verslaggeversblok is. Ook zijn er geen argumenten, dus de waarde van het sleutelwaardepaar **arguments** is een leeg object.

3.3.4 Argument types

Hieronder is de JSON-definitie van het *assert* blok te zien:

```
1 {
2   opcode: 'assert',
3   blockIconURI: feedbackBlockIcon,
4   blockType: BlockType.COMMAND,
5   text: 'assert [ASSERT_CONDITION] named [NAME]',
6   arguments: {
7     ASSERT_CONDITION: {
8       type: ArgumentType.BOOLEAN,
9       defaultValue: false
10    },
11    NAME: {
12      type: ArgumentType.STRING,
13      defaultValue: ''
14    }
15  }
16 }
```

Het resulterende *assert* blok:



Hier focussen we op de argumenten. Het eerste argument is de voorwaarde die gecontroleerd wordt en neemt een Booleaans verslaggeversblok (argumenttype ‘BOOLEAN’), daarom is dit argument zes-hoekig. Het tweede argument is de tekst die moet toegevoegd worden aan de feedbackboom. Het argumenttype is hier ‘STRING’ waardoor het argument aferonde hoeken heeft.

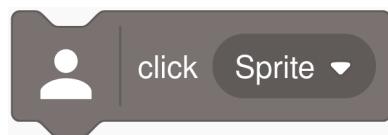
3.3.5 Menu’s

Hieronder is de JSON-definitie van het *click sprite* blok te zien:

```

1  {
2      opcode: 'clickSprite',
3      blockIconURI: interactBlockIcon,
4      blockType: BlockType.COMMAND,
5      text: 'click [SPRITE]',
6      arguments: {
7          SPRITE: {
8              type: ArgumentType.STRING,
9              menu: 'spritesReplaceable'
10         }
11     }
12 }
```

Het resulterende *click sprite* blok:



Hierboven zien we bij de definitie van het argument **SPRITE** dat er een sleutelwaardepaar **menu** is. De waarde (hier **spritesReplaceable**) is een verwijzing naar een menudefinisie. De locatie van de menudefinisie werd kort besproken in Paragraaf 3.2 waar het sleutelwaardepaar **menus** werd aangehaald. Nu zien we hier een voorbeeld van waar **spritesReplaceable** wordt gedefinieerd:

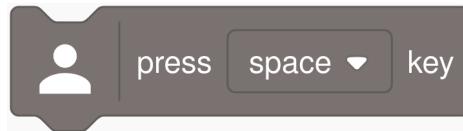
```

1 menus: {
2     spritesReplaceable: {
3         acceptReporters: true,
4         items: '_getSpritesList'
5     }
6 }
```

Het sleutelwaardepaar **acceptReporters** beslist of het menu vervangbaar is of niet. Het *click sprite* blok heeft een vervangbaar menu. Dit zorgt ervoor dat een menu kan vervangen worden door een verslaggeversblok:



Een niet-vervangbaar menu heeft een rechthoekige vorm:



De waarde van het sleutelwaardepaar `items` bevat de naam van een methode die een lijst teruggeeft. Deze lijst bevat de menuwaarden. De methode `_getSpritesList` is gedefinieerd in de uitbreidingsklasse.

3.4 TAKKEN UITVOEREN

De methode `groupName` bevat de functionaliteit van het *test group* blok:

```
1  groupName (args, util) {  
2      // ...  
3      // logica die de feedbackboom uitbreidt.  
4      // ...  
5      util.startBranch(1, true);  
6 }
```

In Paragraaf 3.3 bespraken we het argument `args`. Een functionaliteitsmethode krijgt nog een tweede argument mee: `util`. Het argument `util` bevat info over de Scratchomgeving. Ook bevat het `util` object hulpmethodes zoals `startBranch`. De methode `startBranch` dient om een tak te starten van een C-blok. Dit moet expliciet gedaan worden, anders zullen de ingesloten blokken van een C-blok niet uitgevoerd worden. De methode `startBranch` neemt 2 argumenten:

- `branchNum`: het nummer van de tak die moet gestart worden, startend vanaf 1.
- `isLoop`: moet de functionaliteitsmethode na het uitvoeren van de tak opnieuw uitgevoerd worden?

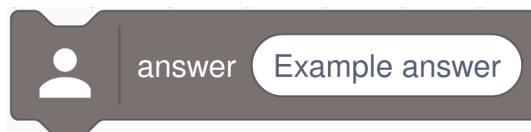
De functionaliteitsmethode van het *test group* blok heeft `isLoop` op `true` staan bij het oproepen van de methode `startBranch`. Dit komt omdat de testgroep na het uitvoeren van de tak moet afgesloten worden. Dit wordt gedaan bij het opnieuw oproepen van de methode `groupName` na het uitvoeren van tak 1. Later gaan we hier dieper op in.

3.5 Wachten

De functionaliteitsmethode van het *answer* blok:

```
1 answer (args, util) {  
2     // if no questions are asked yet, wait for them to be asked  
3     if (this.questions.length === 0) {  
4         util.yield();  
5         return;  
6     }  
7     // answer asked question  
8 }
```

Het resulterende *answer* blok:



Wachten gebeurt met de methode `yield`. Als de methode `yield` wordt opgeroepen zal de functionaliteitsmethode waarin dit wordt opgeroepen, in de toekomst nogmaals uitgevoerd worden. Het *answer* blok antwoord alleen als er een vraag gesteld wordt. Als er geen vraag gesteld wordt zal er gewacht worden tot er een vraag gesteld wordt. Dit kan duidelijk gezien worden in bovenstaand codefragment.

3.6 Conclusie

Het toevoegen van een uitbreiding moet rechtstreeks in de Scratch VM codebase gebeuren. Een uitbreiding kan nieuwe blokken toevoegen aan de Scratchomgeving. Deze blokken kunnen uitbreidings-specifieke dropdownmenu's hebben.

In het volgende hoofdstuk gaan we dieper in op de technieken die gebruikt worden in de functionaliteitsmethodes van de Poke-uitbreiding.

4 Poke: technisch bekeken

De focus van Scratch ligt bij intuïtief kunnen programmeren en constante gebruikersinteractie. Hierdoor is Scratch inherent zeer parallel. Dit maakt het testen van Scratchprojecten uitdagender.

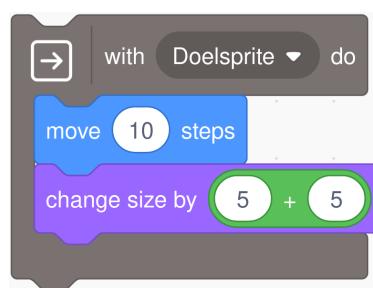
In het vorige hoofdstuk bespraken we algemene functionaliteiten om Scratch uit te breiden. Dit hoofdstuk gaat dieper in op de technische concepten die de Pokeblokken gebruiken in hun functionaliteitsmethodes. We beginnen met de achterliggende werking van UBAS. Ook bekijken we de concepten die de feedbackblokken gebruiken. Daarna zullen we de simulatie van gebruikersinteractie bespreken. En uiteindelijk zien we hoe de waarnemingsblokken werken.

4.1 Uitvoeren van Blokken in een Andere Sprite (UBAS)

In Scratch heeft elke sprite een object dat alle blokken bevat die bij de sprite horen. We noemen dit object de blokmap. Dit object is een map die een blok ID mapt op het corresponderend blok. Tijdens de uitvoering zal de blokmap gebruikt worden om de uit te voeren blokken op te zoeken. Hierdoor kunnen sprites alleen blokken uitvoeren die aanwezig zijn in de blokmap. Als een blok in de Scratchomgeving wordt toegevoegd aan een sprite, zal deze ook aan de blokmap toegevoegd worden.

UBAS zorgt dat een blok dat bij een bronsprite hoort, uitgevoerd wordt in een andere sprite: de doelsprite. We zullen zien dat de implementatie hiervan ingewikkeld wordt, omdat dit concept eigenlijk niet past in het Scratchmodel.

De technische uitleg van UBAS wordt gedaan met een simpel voorbeeld. In Figuur 44 is de Scratchcode te zien die aanwezig is in de bronsprite. Deze code zal simpelweg de doelsprite bewegen en vergroten. De Scratchcode die moet uitgevoerd worden in de doelsprite noemen we de payload-code.



Figuur 44: Scratchcode aanwezig in de bronsprite.

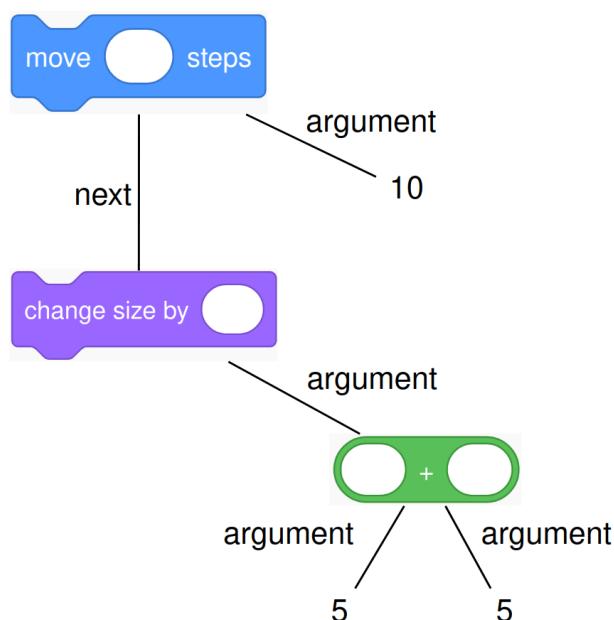
De payload-code uitvoeren in de doelsprite is niet triviaal, want deze blokken zijn niet aanwezig in de blokmap van de doelsprite. De Poke-uitbreiding zal de volgende stappen ondernemen om de payload-code te kunnen uitvoeren in de doelsprite:

1. Voeg de payload-code toe aan de blokmap van de doelsprite: dit noemen we blokinjectie.

2. Voeg een *when broadcast received* blok toe dat de payload-code kan starten.
3. Start het *when broadcast received* blok.

4.1.1 Blokinjectie

Scratchcode is geschreven volgens een boomstructuur. Figuur 45 toont de boomstructuur van de payload-code. Deze boom heeft 6 toppen, die elk hun eigen sleutelwaardepaar voorstellen in de blokmap. Merk op dat zelf stringwaardes (zoals “10”) aparte toppen zijn in de boom.



Figuur 45: De boomvoorstelling van de te injecteren Scratchcode.

Het eerste dat we doen bij UBAS is alle Scratchcode in de bronsprite (de testsprite) injecteren in de doelsprite. We injecteren direct alle testcode omdat dit technisch gemakkelijker is dan manueel de boom te overlopen en blokken te selecteren voor injectie. We kopiëren dus de volledige blokmap van de bronsprite, en voegen alle sleutelwaardeparen ervan toe aan de blokmap van de doelsprite.

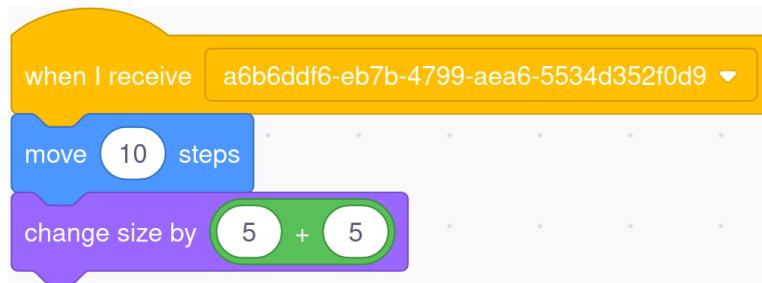
Blokinjectie zorgt niet voor een visuele update in de Scratch GUI. De payload-code zal dus niet zichtbaar zijn in de doelsprite.

Poke houdt intern bij waar de testcode geïnjecteerd werd. Bij het starten van de testcode worden alle injecties die gedaan werden bij de vorige uitvoering van de testcode ongedaan gemaakt.

Na blokinjectie is de payload-code aanwezig in de doelsprite, maar er is nog geen manier om deze uit te voeren.

4.1.2 Het *when broadcast received* blok toevoegen

De geïnjecteerde payload-code moet uitgevoerd worden vanuit de functionaliteitsmethode van het UBAS-blok (hier het *with sprite do* blok). Het UBAS-blok bevindt zich bovendien in de bronsprite. We voegen een *when broadcast received* blok toe aan de doelsprite. Het *when broadcast received* blok zal als volgende blok het eerste blok van de payload-code hebben (zie Figuur 46).



Figuur 46: De payload-code die kan uitgevoerd worden door een *when broadcast received* te activeren.

Het signaal is een UUID. Dit voorkomt interferentie van test- en programmacode.

4.1.3 Het *when broadcast received* blok starten

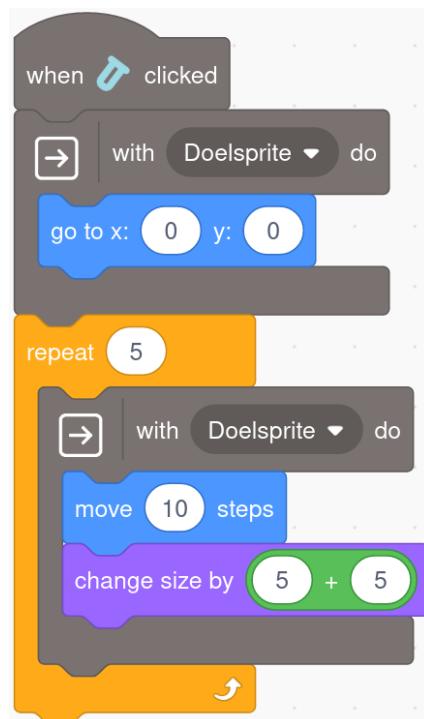
Nu kunnen we simpelweg het *when broadcast received* startblok met het UUID signaal dat bij de payload-code hoort starten. Dit doen we met de methode `startHats` die beschikbaar is in het `util` object dat meegegeven wordt aan elke functionaliteitsmethode (zie Hoofdstuk 3).

```
1 startedThreads = this.runtime.startHats(  
2   'event_whenbroadcastreceived', {  
3     BROADCAST_OPTION: "a6b6ddf6-eb7b-4799-aea6-5534d352f0d9"  
4   }  
5 );
```

Merk op dat we hiervoor geen signaal verzenden. Dankzij de bestaande methode `startHats` kunnen we rechtstreeks het juiste startblok starten. Dit zorgt ervoor dat we gemakkelijk kunnen wachten op de gestarte threads omdat we deze terugkrijgen van de methode `startHats` (en opslaan in `startedThreads` hierboven). Het verhindert ook dat we *broadcast* blokken in de testcode moeten injecteren.

4.1.4 UBAS efficiënt maken

We bekijken een iets ingewikkelder voorbeeld om aan te tonen dat er zuinig wordt omgesprongen met het injecteren van testcode en het toevoegen van *when broadcast received* blokken. Figuur 47 toont testcode die 2 UBAS-blokken bevat, en tot 6 UBAS-blokuitvoeringen zal leiden tijdens uitvoering.



Figuur 47: Testcode die meerder uitvoeringen van UBAS-blokken zal hebben.

De uitvoering van het eerste UBAS-blok zal:

1. Alle testcode injecteren in de doelsprite.
2. Een *when broadcast received* blok met signaal *s1* toevoegen in de doelsprite. Dit blok zal als volgende blok het *go to* blok hebben.
3. Het *when broadcast received* blok met signaal *s1* starten.

Daarna zal het tweede UBAS-blok 5 maal uitgevoerd worden. De testcode was al eerder geïnjecteerd in de doelsprite, dus dit wordt niet opnieuw gedaan. In de eerste iteratie van de lus zal het UBAS-blok:

1. Een *when broadcast received* blok met signaal *s2* toevoegen in de doelsprite. Dit blok zal als volgende blok het *move steps* blok hebben.
2. Het *when broadcast received* blok met signaal *s2* starten.

In de 4 resterende iteraties zal alleen het *when broadcast received* blok met signaal *s2* gestart worden.

We zien dus dat er nooit dubbel geïnjecteerd wordt, en nooit dubbele *when broadcast received* blokken worden aangemaakt.

Technisch doen we dit met een aantal datastructuren:

- `testCodeInjected`: een lijst die de id's bevat van alle sprites waar de testcode al in geïnjecteerd is. Deze lijst wordt gebruikt om te controleren of de testcode nog moet geïnjecteerd worden.
- `injecterBlockIdToInjectedSpriteIdToBroadcastMessage`: een 2d map die het signaal bijhoudt gegeven de id van het UBAS-blok en de id van de doelsprite (UBAS-blok id -> doelsprite id -> signaal). Deze datastructuur wordt gebruikt om te controleren of een *when broadcast received* blok nog moet toegevoegd worden. Dit wordt ook gebruikt om het signaal te weten te komen bij het starten van een eerder toegevoegd *when broadcast received* blok.

4.1.5 De UBAS-blokken

De functionaliteitsmethodes van het *with sprite do* blok en het *sprite filter* blok worden besproken.

4.1.5.1 Het *with sprite do* blok

Dit is een vereenvoudigde versie van de functionaliteitsmethode van het *with sprite do* blok:

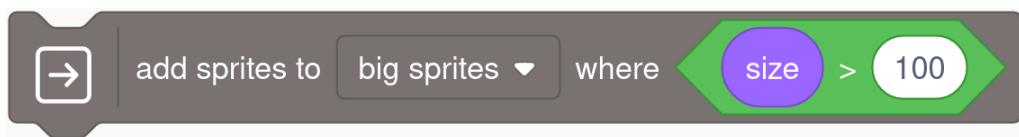
```
1 withSpriteDo (args, util) {  
2     this._injectAllCodeIfNeeded(...);  
3  
4     if (!this._broadcastReceivedBlockIsPresent(...)) {  
5         this._injectBroadcastThread(...);  
6     }  
7  
8     this._runInjectedBroadcastThreadsIfNeeded(...);  
9  
10    this._waitForStartedThreads(...);  
11 }
```

Lijn 2 injecteert alle testcode in de doelsprite als dit nog niet eerder gedaan werd. Het if-statement op lijn 4 voegt het *when broadcast received* blok toe dat bij de payload-code hoort als dit nog niet eerder gedaan werd. Op lijn 8 wordt het *when broadcast received* blok uitgevoerd als dit nog niet eerder gedaan werd. Na het uitvoeren van het *when broadcast received* blok wordt er gewacht (lijn 10) tot de payload-code klaar is met uitvoeren. De methode `_waitForStartedThreads` is niet blokkerend. Deze methode zal simpelweg `yield` oproepen als er nog verder moet gewacht worden in de testcode. Later zal de functionaliteitsmethode opnieuw opgeroepen worden en zal `_waitForStartedThreads` opnieuw uitgevoerd worden. Als uiteindelijk de functionaliteitsmethode opgeroepen wordt en ten einde loopt zonder dat `yield` werd opgeroepen, zal de uitvoering

van de testcode verder gaan. Het is dus de thread die de testcode uitvoert die zal wachten. Andere threads (bijvoorbeeld programmacode threads) voeren verder uit.

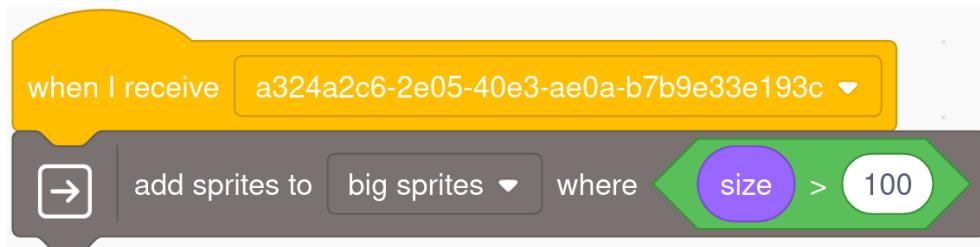
4.1.5.2 Het *sprite filter* blok

Het *sprite filter* blok (Figuur 48) heeft een voorwaarde die uitgevoerd wordt in alle sprites. Als de voorwaarde in een sprite naar **true** evalueert, wordt de naam van de sprite toegevoegd aan de geselecteerde lijst.



Figuur 48: Het *sprite filter* blok.

Vanuit technisch standpunt moet er dus een injectie gedaan worden van de voorwaarde in elke sprite, en daar moet de voorwaarde geëvalueerd worden. Het probleem hiermee is dat de voorwaarde een verslaggeversblok is. Een *when broadcast received* blok is een commandoblok en kan alleen gekoppeld worden aan een ander commandoblok, niet aan een verslaggeversblok. In elke doelsprite moeten we dus een *when broadcast received* blok hebben, gevolgd door een ander commandoblok, dat als argument de voorwaarde bevat. Dit is te zien in Figuur 49.



Figuur 49: De voorwaarde die we willen evalueren in de doelsprite ingesloten in een *sprite filter* blok.

We gebruiken het *sprite filter* blok om de voorwaarde te kunnen evalueren in de doelsprite. Zo kunnen we ook de functionaliteitsmethode van het *sprite filter* blok gebruiken voor de evaluatie.

Een eenvoudigere versie van de functionaliteitsmethode is hier te zien:

```

1  spriteFilter (args, util) {
2      if (
3          // Als de functionaliteitsmethode uitgevoerd wordt door een geï
4          njecteerd sprite filter blok
5      ) {
```

```

5      this.
6          injectedBooleanBlockIdToInjectedSpriteIdToBooleanBlockResult
7              [booleanStatementBlockId][targetSpriteId] = args.CONDITION;
8      return;
9  }
10
11 // Code vanaf hier wordt alleen uitgevoerd door sprite filter
12 // blokken die niet geïnjecteerd zijn.
13
14 const sprites = ...;      // Alle sprites
15 const broadcastMessage = uid();
16 for (const doelsprite of sprites) {
17     // Injecteer alle testcode in de doelsprite als dit nog niet
18     // eerder gedaan werd.
19     this._injectAllCodeIfNeeded(...);
20
21     // Voeg het when broadcast received blok toe dat bij de payload
22     // -code hoort als dit nog niet eerder gedaan werd.
23     if (!this._broadcastReceivedBlockIsPresent(...)) {
24         this._injectBroadcastThread(..., broadcastMessage);
25     }
26 }
27
28 this._runInjectedBroadcastThreadsIfNeeded(...);
29 }
```

Lijn 11-21 zal voor elke sprite:

1. de testcode injecteren (als dit niet eerder gedaan werd)
2. het *when broadcast received* blok toevoegen (als dit niet eerder gedaan werd)

Merk op dat de toegevoegde *when broadcast received* blokken allemaal hetzelfde signaal hebben waardoor later met 1 oproep van de methode **startHats** allemaal samen kunnen uitgevoerd worden. Dit wordt gedaan in lijn 23.

De functionaliteitsmethode wordt ook uitgevoerd als het *sprite filter* blok geïnjecteerd werd. Een uitvoering van de functionaliteitsmethode voor een *sprite filter* blok zal simpelweg leiden tot het opslaan van het resultaat van de geëvalueerde voorwaarde (lijn 2-7). Hiervoor wordt een datastructuur gebruikt: **injectedBooleanBlockIdToInjectedSpriteIdToBooleanBlockResult**. Dit is een 2d map die het resultaat van een Booleaanse voorwaarde bijhoudt voor elke sprite waar deze werd uitgevoerd (Booleaans verslaggeversblok id -> sprite -> id -> resultaat).

Eens alle geïnjecteerde *sprite filter* blokken klaar zijn met uitvoeren, wordt met de opgeslagen resultaten de gefilterde lijst van sprites gemaakt (lijn 25-28).

4.1.6 Discussie

De implementatie van de UBAS-blokken misbruikt het *when broadcast received* blok. Initieel was het plan om werkelijk een broadcast te sturen, maar dit zou de implementatie technisch moeilijker maken dan nodig. Een betere oplossing zou geweest zijn om een nieuw startblok te definiëren in de Poke-uitbreiding, dat als enige doel heeft om gebruikt te worden bij het starten van geïnjecteerde code. Net zoals het signaal in het *when broadcast received* startblok zou dit nieuwe startblok, ook een string-waarde als argument moeten hebben waar een UUID wordt in opgeslagen. Dit zodat elke instantie van het startblok apart kan opgeroepen worden via de methode `startHats`. Het nieuw startblok moet dan ook onzichtbaar gemaakt worden in het uitbreidingsmenu zodat de leerkracht deze niet te zien krijgt.

4.2 Feedback geven

Om feedback te geven, wordt tijdens het testen de feedbackboom opgebouwd. De feedbackboom wordt weergegeven in de Scratch GUI. Daar voegden we code toe die de feedbackboom kan visualiseren.

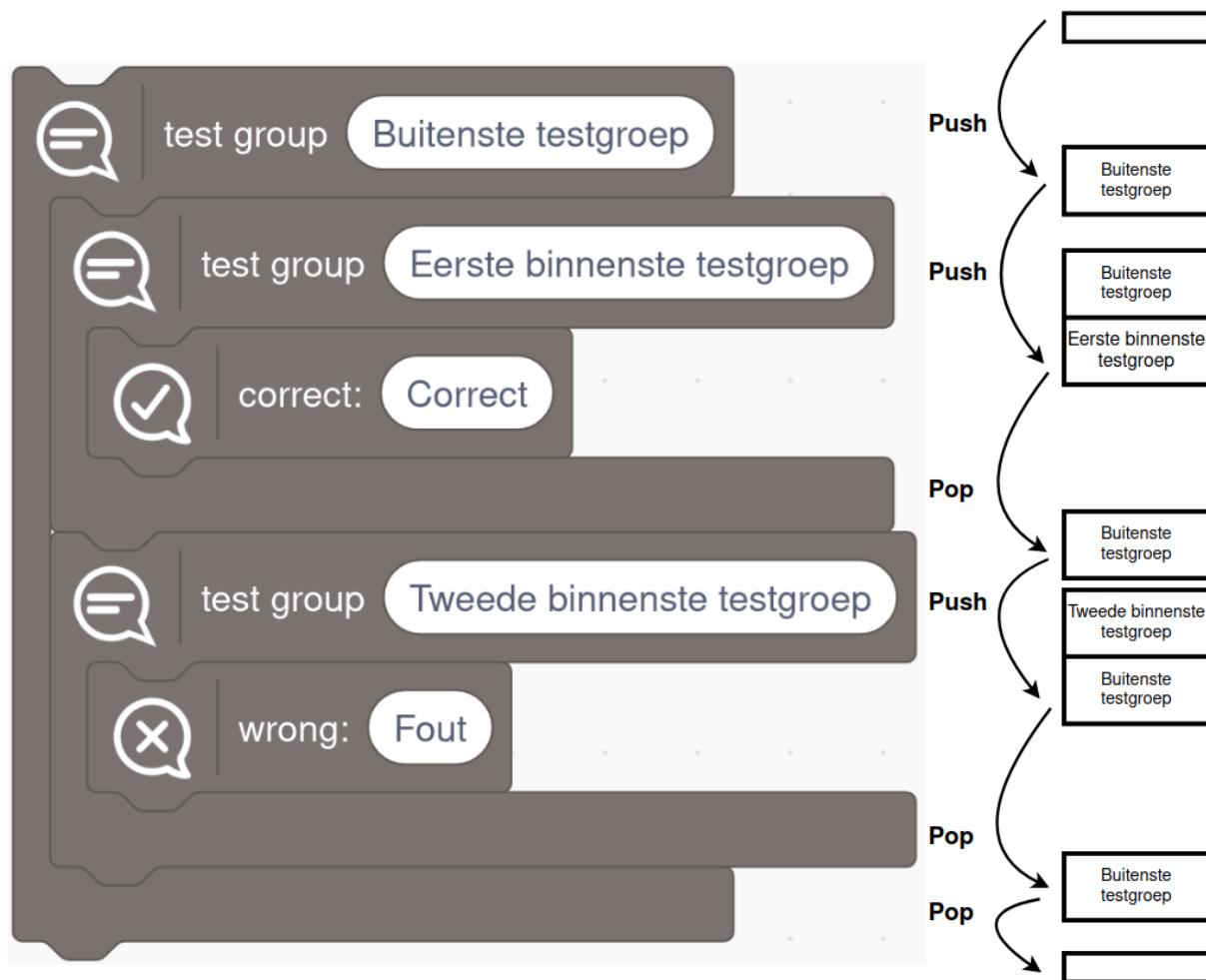
In de functionaliteitsmethodes van de feedbackblokken bouwen we de feedbackboom op. We kunnen toppen toevoegen die tekst bevatten en een status die aangeeft of ze als gefaalde top moeten weergegeven worden of niet.

Hoe een feedbackboom wordt opgebouwd, wordt best uitgelegd aan de hand van een voorbeeld. Figuur 50 toont testcode die een feedbackboom opbouwt. Figuur 51 toont de resulterende feedbackboom.

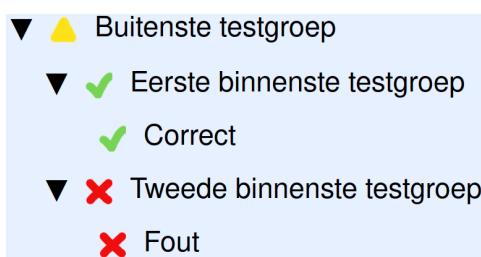
Waar we tijdens de opbouw zitten in de feedbackboom wordt bijgehouden met een stack: de parsestack. Figuur 50 toont ook een visualisatie van de evolutie van de parsestack tijdens het uitvoeren van de testcode. Daar kunnen we duidelijk zien dat een *test group* blok telkens zichzelf pusht op de parsestack net voor de uitvoering van zijn ingesloten blokken. Bij het verlaten van een *test group* blok wordt hij terug gepopt van de parsestack. Het onderste element van de stack is dus telkens de testgroep waarin we zitten.

Een *test group* blok zal ook telkens een top toevoegen aan de boom. De ouder van een nieuwe top is altijd de top die onderaan op de parsestack staat, ofwel de testgroep waar we inzitten.

We nemen als voorbeeld het uitvoeren van het *test group* blok met feedback “Eerste binnenste testgroep”. Net voor dit blok wordt uitgevoerd staat alleen “Buitenste testgroep” op de parsestack. En we



Figuur 50: Testcode met de stackoperaties dat de feedbackblokken doen op de parsestack. De parsestack groeit naar beneden.



Figuur 51: De feedbackboom dat bij de testcode van Figuur 51 hoort.

zien in de uiteindelijke feedback (Figuur 51) dat “Eerste binnenste testgroep” ook werkelijk een kind van “Buitenste testgroep” is. Aan het einde van het *test group* blok met feedback “Eerste binnenste testgroep” wordt “Eerste binnenste testgroep” van de parsestack gepopped. Dit is logisch, want vanaf dan willen we niet meer dat nieuwe toppen als kind van “Eerste binnenste testgroep” toegevoegd worden.

4.2.1 Het *test group* blok

Dit is een vereenvoudigde versie van de functionaliteitsmethode van het *test group* blok:

```
1  testGroup (args, util) {
2      // Als de onderste top op de parsestack hetzelfde is als het blok
      // dat dit uitvoert,
3      // dan worden we voor de tweede keer uitgevoerd, wat betekent dat
      // we het C-blok verlaten
4      if (tree.peekParseStack().blockId === this._getCurrentBlockId(util))
5          {
6              // Pop de top
7              tree.getParseStack().pop();
8
9          } else {
10             // Voeg de top van deze testgroep toe aan de boom (en
11             // automatisch aan de parsestack).
12             tree.addChild(this._getCurrentBlockId(util), args.GROUP_NAME);
13
14             // Start de ingesloten blokken (branch: 1) en voer deze methode
15             // nog eens uit als ze klaar zijn (loop: true).
16             util.startBranch(1, true);
17         }
18     }
```

Het if-else statement splitst de functionaliteitsmethode op. Het “else” deel wordt uitgevoerd bij het openen van de testgroep en zal daarom de testgroep toevoegen aan de feedbackboom. Dit zal automatisch de testgroeptop pushen op de parsestack (lijn 11). Ook zal het de ingesloten blokken starten met de methode `startBranch` (lijn 14). Het argument met waarde `true` van de methode `startBranch` geeft aan dat de functionaliteitsmethode opnieuw moet opgeroepen worden als deze blokken klaar zijn met uitvoeren. Bij het opnieuw oproepen van de functionaliteitsmethode zal het “if” deel worden uitgevoerd. Dit sluit de testgroep door testgroeptop van de parsestack te verwijderen (lijn 6).

4.2.2 Het assert blok

Tot nu toe bespraken we interne toppen. Een blad toevoegen aan de feedbackboom wordt gelijkaardig gedaan. Dit is een vereenvoudigde versie van de functionaliteitsmethode van het *assert* blok:

```

1 assert (args, util) {
2     tree.addChild(this.getCurrentBlockId(util), args.NAME);
3     if (!args ASSERT_CONDITION) {
4         tree.peekParseStack().setFailed();
5     }
6     tree.getParseStack().pop();
7 }
```

Op lijn 2 voegen we een top toe aan de boom (en automatisch aan de parsestack). Daarna zetten we die top op gefaald als de “ASSERT_CONDITION” **false** is (lijn 3-5). Uiteindelijk halen we de top direct terug van de parsestack omdat we een blad aan het toevoegen zijn (lijn 6).

4.2.3 Het wait until or timeout blok

Bij het implementeren van het *wait until or timeout* blok maken we gebruik van `util.stackFrame` om bij te houden wanneer de eerste oproep van de functionaliteitsmethode was. Het `util.stackFrame` kan gebruikt worden om tijdens de uitvoering van een blok, doorheen de verschillende oproepen van de functionaliteitsmethode, informatie bij te houden. Ook wordt gebruik gemaakt van `runtime.currentMSecs` waarde om de tijd te achterhalen.

Dit is de functionaliteitsmethode van het *wait until or timeout* blok:

```

1 waitUntilOrTimeout (args, util) {
2     if (!util.stackFrame.startTime) {
3         util.stackFrame.startTime = this.runtime.currentMSecs;
4     }
5
6     if (args.CONDITION) {
7         util.startBranch(1, false);
8         return;
9     }
10
11    if (this.runtime.currentMSecs - util.stackFrame.startTime > args.
12        SECONDS * 1000) {
13        util.startBranch(2, false);
14        return;
15    }
16    util.yieldTick();
17 }
```

Het eerste if-statement controleert of `util.stackFrame.startTime` bestaat. Als dit niet zo is, zitten we in de eerste oproep van de functionaliteitsmethode tijdens een uitvoering van het *wait until or timeout* blok. Daarom stellen we de starttijd in.

Het tweede if-statement zal de blokken in de eerste tak van het *wait until or timeout* blok uitvoeren als de voorwaarde `true` is.

Het derde if-statement zal controleren of de tijdslimiet overschreden is. Als dit zo is, worden de blokken in de tweede tak uitgevoerd.

Uiteindelijk, als de voorwaarde nog niet `true` is en de tijdslimiet is nog niet overschreden, dan wordt er gewacht (lijn 16) en zal de volledige functionaliteitsmethode dus later nog eens opgeroepen worden.

4.2.4 Discussie

De implementatie van het *test group* blok kan waarschijnlijk gemakkelijker door gebruik te maken van de `util.stackFrame`. Momenteel wordt de `parsestack` gebruikt om bij te houden of de functionaliteitsmethode van het *test group* blok voor de eerste keer of voor de tweede keer wordt opgeroepen tijdens een uitvoering van het *test group* blok. Maar zoals we zagen in Paragraaf 4.2.3 kan `util.stackFrame` hier ook voor gebruikt worden. Zo zou de `parsestack` dus niet nodig zijn.

4.3 Gebruikersinteractie simuleren

Simulatie van gebruikersinteractie wordt op verschillende manieren gedaan. Gebruikersinteractie die startblokken activeert, is het gemakkelijkst om te simuleren.

4.3.1 Startblokken activeren en er op wachten

Testblokken zoals *press green flag* zijn gemakkelijk te implementeren. De functionaliteitsmethode gebruikt simpelweg de methode `runtime.startHats`:

```
1 pressGreenFlag () {  
2     this.runtime.startHats('event_whenflagclicked');  
3 }
```

Het *press key* blok en *click sprite* blok werken gelijkaardig. De functionaliteitsmethode van het *press key* blok activeert 2 groepen startblokken: de startblokken die activeren bij het indrukken van een specifieke toets en de startblokken die activeren bij het indrukken van eender welke toets:

```

1 pressKey (args) {
2     this.runtime.startHats('event_whenkeypressed', {
3         KEY_OPTION: args.KEY
4     });
5     this.runtime.startHats('event_whenkeypressed', {
6         KEY_OPTION: 'any'
7     });
8 }

```

Het *press key and wait* blok en het *click sprite and wait* blok gebruiken ook de methode `runtime.startHats` om threads op te starten. Daarbovenop gebruiken ze ook nog eens de methode `util.yield` om te wachten op de gestarte threads.

4.3.2 De muisaanwijzer bewegen

Het bewegen van de muisaanwijzer kan niet gesimuleerd worden door het activeren van startblokken. Hier simuleren we exact hoe de Scratch GUI de Scratch VM zou aanspreken. We gebruiken het `ioDevices` object om muisaanwijzerinfo door te geven.

De methode `ioDevices.mouse.postdata` neemt coördinaten in het browsercoördinaatstelsel. Het *move mouse to* blok neemt coördinaten in het Scratchcanvascoördinaatstelsel. Daarom wordt eerst een conversie gedaan.

```

1 moveMouseTo (args, util) {
2     const clickData = {};
3     clickData.canvasWidth = util.runtime.renderer.canvas.width;
4     clickData.canvasHeight = util.runtime.renderer.canvas.height;
5     clickData.x = ((args.X + 240) * clickData.canvasWidth) / 480;
6     clickData.y = ((args.Y - 180) * clickData.canvasHeight) / -360;
7     util.runtime.ioDevices.mouse.postData(clickData);
8 }

```

Het eerste dat de methode `postData` doet, is de coördinaten terug omzetten naar Scratchcanvascoördinaten:

```

1 postData (data) {
2     if (data.x) {
3         this._clientX = data.x;
4         this._scratchX = Math.round(MathUtil.clamp(
5             480 * ((data.x / data.canvasWidth) - 0.5),
6             -240,
7             240
8         ));
9     }
10    if (data.y) {
11        this._clientY = data.y;

```

```

12     this._scratchY = Math.round(MathUtil.clamp(
13         -360 * ((data.y / data.canvasHeight) - 0.5),
14         -180,
15         180
16     ));
17 }
18 ...
19 }
```

4.3.3 De vragenwachtlijst

Het *answer* blok kan antwoorden op gestelde vragen. Als een Scratchvraag gesteld wordt, zal een Scratchgebeurtenis “QUESTION” opgegooid worden. In de constructor wordt een callback gedefinieerd bij het opgooien van deze gebeurtenis:

```

1 class Scratch3PokeBlocks {
2     constructor (runtime) {
3         ...
4         this.questions = [];
5         this.runtime.on('QUESTION', question => {
6             this.questions.push(question);
7         });
8         ...
9     }
10 }
```

Dit zorgt dat een FIFO-wachtlijst van vragen wordt opgesteld. Als het *answer* blok uitgevoerd wordt, zal de eerste vraag in de lijst beantwoord worden. Antwoorden op een vraag wordt ook gedaan met een Scratchgebeurtenis: “ANSWER”. Als er geen vragen in de wachtlijst staan, zal het *answer* blok wachten.

```

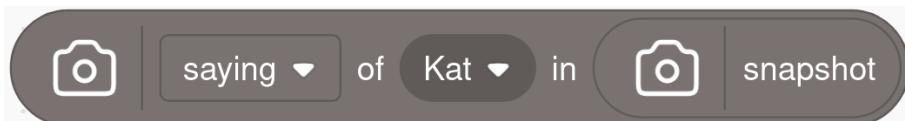
1 answer (args, util) {
2     // Als er geen vragen in de wachtlijst staan om beantwoord te
3     // worden, wacht op een vraag.
4     if (this.questions.length === 0) {
5         util.yieldTick();
6     }
7     // Beantwoord de vraag
8     this.runtime.emit('ANSWER', args.ANSWER);
9     this.questions.shift();
```

4.4 Waarnemingen doen

Het *snapshot* blok neemt een snapshot van de omgeving. Achterliggend zal dit verslaggeversblok een JSON-object in stringvorm teruggeven. Dit JSON-object is een lijst van alle sprites. Elke sprite heeft een aantal eigenschappen waarvan de waarde werd opgeslagen. Deze eigenschappen werden eerder besproken in Paragraaf 2.4.2.

Bij het bevragen van een snapshot zal de functionaliteitsmethode van het *query snapshot* blok de stringvorm van het JSON-object meekrijgen. Deze wordt terug omgezet naar een JSON-object zodat hier informatie kan uitgehaald worden.

In Figuur 52 zal het *snapshot* blok dus een JSON-object maken, deze omzetten naar een string en uiteindelijk deze string doorgeven aan het *query snapshot* blok. Het *query snapshot* blok zal deze string terug omzetten naar een JSON-object. Daarna kan het JSON-object gebruikt worden om te achterhalen wat de sprite “Sprite” aan het zeggen was toen de snapshot genomen werd.



Figuur 52: Een combinatie van het *query snapshot* blok en het *snapshot* blok.

4.5 Conclusie

In dit hoofdstuk werden de technische concepten uitgelegd die de Pokeblokken gebruiken in hun functionaliteitsmethodes. We merken ook direct dat UBAS, een functionaliteit die het Scratchmodel breekt, moeilijk te implementeren is, terwijl gebruikersinteractie simuleren door startblokken te activeren zeer gemakkelijk kan. Ook bekijken we hoe de feedbackboom opgesteld wordt en hoe we verslaggeversblokken die strings teruggeven kunnen gebruiken om JSON-objecten door te geven.

5 Validatie

In dit hoofdstuk testen we de oefeningen die werden gebruikt in de jeugdrondes van de Vlaamse Programmeerwedstrijd in 2017 als validatie van de Poke-uitbreiding. Dit betekend dat de uitleg die bij de oefening hoort niet zelf zijn opgesteld. De opgaven van de oefeningen zijn overgenomen van de Itchmasterproef (Mak, Dawyndt, en Scholliers 2019). Voor alle oefeningen maakten we een extra sprite. Deze sprite bevat telkens alle testcode en is onzichtbaar op het Scratchcanvas.

De oefeningen zijn op te delen in een aantal categorieën:

	Uitleg categorie	Oefeningen
Praten	Sprites praten, denken of vragen zaken.	Op bezoek bij Devin, flauw mopje, cijfersom
Bewegende sprites	Sprites veranderen van locatie.	Heksenjacht, voetballende kat, vliegende papegaai
Animerende sprites	Sprites veranderen van uiterlijk. Dit wordt vaak gedaan om sprites te laten animeren.	Mad hatter, voetballende kat, vliegende papegaai
Spelletjes	Een Scratchoefening dat constante gebruikersinteractie nodig heeft.	Vang de appels, pong (appendix A)
Tekenen	De Penuitbreiding wordt gebruikt om zaken te tekenen.	Teken een vierkant, teken een driehoek, teken een huis

De oefening “pong” maakt geen deel uit van de Vlaamse Programmeerwedstrijd opgaven, maar hebben we wel getest. De testcode is te vinden in de appendix A. Nu volgt de bespreking van de oefeningen. Voor elke oefening tonen we de opgave die er bij hoort, geven we een modeloplossing en bespreken we een mogelijk testplan met de bijhorende feedback. Dit hoofdstuk zal aantonen voor welke categorieën van oefeningen Poke kan gebruikt worden.

5.1 Op bezoek bij Devin

Opgave

Je bent te gast in Devin zijn kamer, en hij wil ontzettend graag kennis met je maken. Schrijf een programma dat:

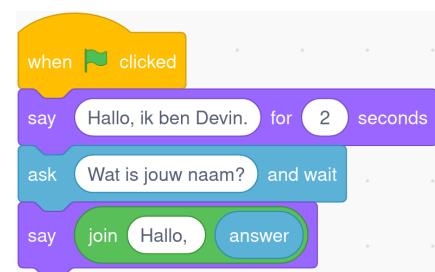
- begint als je op de groene vlag drukt
- Devin laat zeggen “Hallo, ik ben Devin.”
- Devin jouw naam laat vragen
- Devin laat zeggen “Hallo, ...”, waarbij jouw naam wordt ingevuld op de plaats van de drie puntjes

Tip: De eerste twee taken zijn al geprogrammeerd. De rest van het programma kan je maken met de blokken die klaarstaan.

In deze oefening zal de sprite “Devin” zaken zeggen en vragen. Hierdoor hoort de oefening thuis in de categorie “praten”. Het Scratchcanvas dat bij deze oefening hoort is te zien in Figuur 53a. Een modeloplossing is gegeven in Figuur 53b.



(a) Het Scratchcanvas



(b) Een voorbeeldoplossing die zich bevindt in de sprite ”Devin”.

Figuur 53: De oefening ”op bezoek bij Devin”.

Onze testcode (Figuur 54) doet het volgende:

1. Klik op de groene vlag.
2. Controleer dat er “Hallo, ik ben Devin” gezegd wordt.
3. Controleer dat er “Wat is jouw naam?” gezegd wordt:
 - a) Als het vorige correct is, antwoord “Iwijn”
 - b) Controleer dat er “Hallo Iwijn” gezegd wordt.
4. Stop programma- en testcode.

We bekijken de feedback die de testcode genereert bij verschillende oplossingen.

Bij de modeloplossing:

- ✓ Devin zegt "Hallo, ik ben Devin"
- ✓ Devin vraagt naar mijn naam.
- ✓ Devin zegt: "Hallo, Iwijn" nadat "Iwijn" als naam werd geantwoord.

Als Devin je niet groet:

- ✗ Devin zegt geen: "Hallo, ik ben Devin"
- ✓ Devin vraagt naar mijn naam.
- ✓ Devin zegt: "Hallo, Iwijn" nadat "Iwijn" als naam werd geantwoord.

Als Devin je groet, maar niet naar je naam vraagt:

- ✓ Devin zegt "Hallo, ik ben Devin"
- ✗ Devin vraagt niet naar mijn naam!

Merk hier op dat er minder feedback is. Als er niet gevraagd wordt naar een naam moet er ook niet gecontroleerd worden dat Devin hallo zegt.

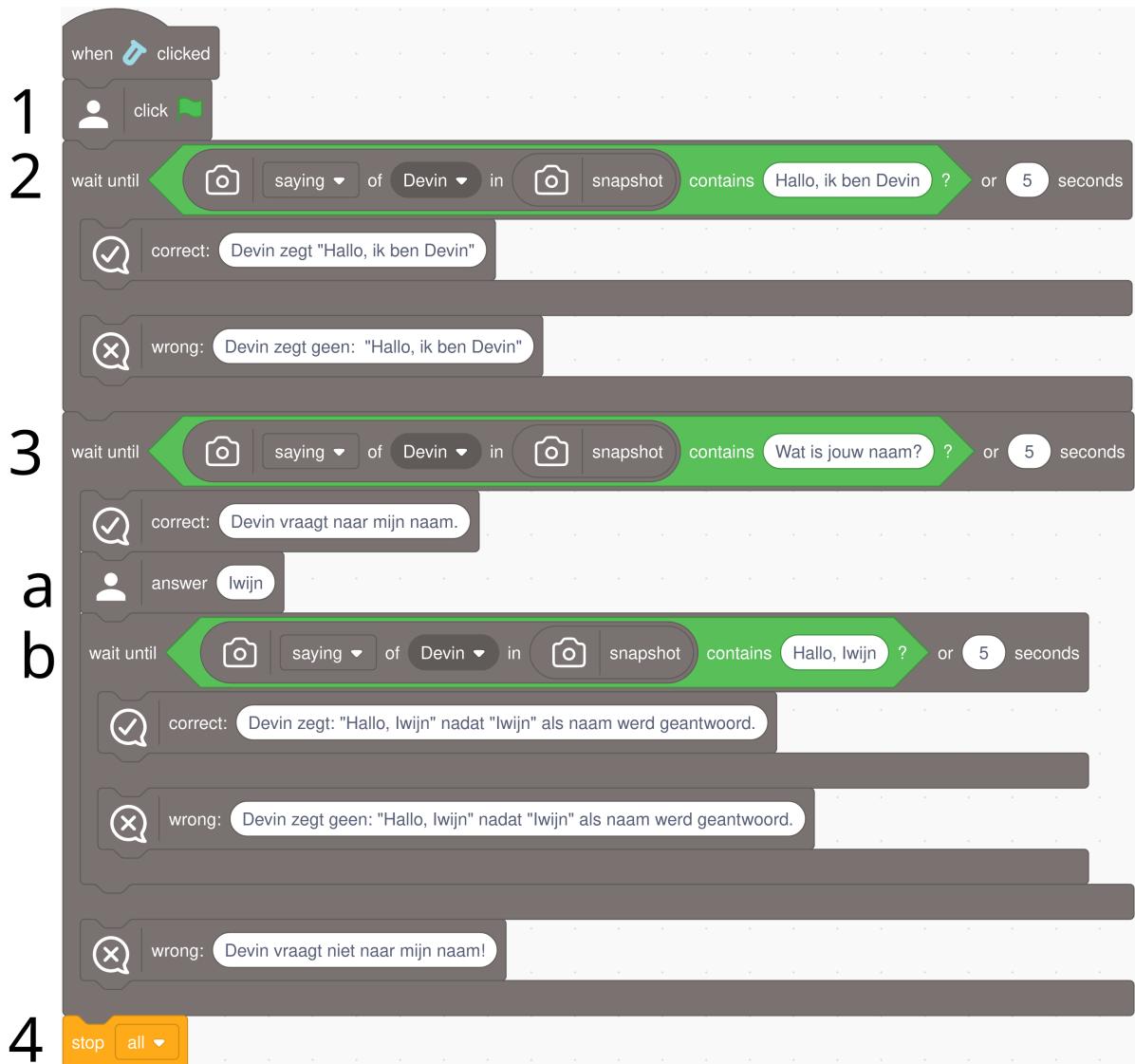
Als Devin je groet, naar je naam vraagt, maar geen hallo zegt:

- ✓ Devin zegt "Hallo, ik ben Devin"
- ✓ Devin vraagt naar mijn naam.
- ✗ Devin zegt geen: "Hallo, Iwijn" nadat "Iwijn" als naam werd geantwoord.

Poke kan momenteel geen onderscheid maken tussen iets dat gezegd wordt met een *ask* blok en iets dat gezegd wordt met een *say* blok. Deze functionaliteit toevoegen aan Poke is mogelijk en is geen technische beperking van Scratch. Hierdoor zal de tweede controle slagen als Devin "Wat is jouw naam?" zegt met een *say* blok. Omdat er niets gevraagd wordt, zal het *answer* blok oneindig lang wachten om te antwoorden. Dit kan opgelost worden door ook aan het *answer* blok een time-out toe te voegen, of door het mogelijk te maken om met het *query snapshot* blok "asking" op te kunnen vragen.

Stap 4 stopt de test- en programmacode. Dit is hier niet nodig, maar het lijkt ons een goed idee om na het testen alle threads sowieso te stoppen.

We zien bij deze oefening dat Poke kan gebruikt worden om de belangrijkste delen van de opgave te testen.



Figuur 54: Testcode voor de oefening “op bezoek bij Devin”.

5.2 Flauw mopje

Opgave

Op het speelveld staan een kat en een tros bananen. Schrijf een programma dat de kat en de bananen volgende (flauwe) mop laat vertellen:

- laat de kat voor 2 seconden zeggen “Klop, klop!”
- laat de bananen daarna voor 2 seconden zeggen “Wie is daar?”
- laat de kat daarna voor 2 seconden zeggen “Ki!”
- laat de bananen daarna voor 2 seconden zeggen “Ki, wie?”
- laat de kat daarna voor 2 seconden zeggen “Neen, dank je. Ik heb liever bananen!”

Het programma mag pas beginnen wanneer op de groene vlag wordt geklikt.

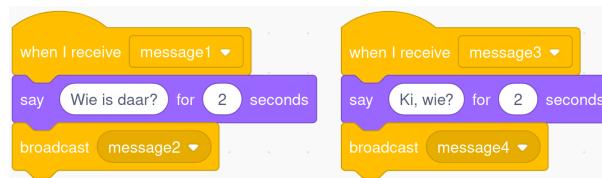
In deze oefening zullen de sprites “Bananen” en “Kat” zaken zeggen. Hierdoor hoort de oefening thuis in de categorie “praten”. Het Scratchcanvas dat bij deze oefening hoort, is te zien in Figuur 55. Een modeloplossing voor de sprites “Bananen” en “Kat” is gegeven in respectievelijk Figuur 56 en Figuur 57. Daar zien we dat er na het indrukken van de groene vlag “Klop, klop!” zal gezegd worden voor 2 seconden. Daarna wordt een signaal verstuurt dat code in de sprite “Bananen” zal starten. Deze code zal opnieuw iets zeggen en een signaal versturen dat dan weer code in de sprite “Kat” zal starten. Dit gaat zo door tot alles gezegd is.



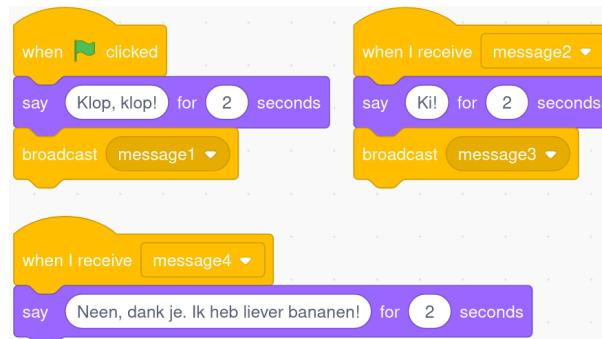
Figuur 55: Het Scratchcanvas van de oefening “flauw mopje”.

Onze testcode (Figuur 58) doet het volgende:

1. Klik op de groene vlag.
2. Controleer dat de kat “Klop, klop!” zegt.



Figuur 56: De voorbeeldoplossing van de oefening “flauw mopje” voor de sprite “Bananen”.



Figuur 57: De voorbeeldoplossing van de oefening “flauw mopje” voor de sprite “Kat”.

3. Controleer dat de bananen “Wie is daar?” zeggen.
4. Controleer dat de kat “Ki!” zegt.
5. Controleer dat de bananen “Ki, wie?” zeggen.
6. Controleer dat de kat “Nee, dank je. Ik heb liever bananen!” zegt.
7. Stop programma- en testcode.

De feedback die deze testcode genereert bij de modeloplossing is als volgt:

- ✓ De kat zegt "Klop, klop!"
- ✓ De bananen antwoorden "Wie is daar?"
- ✓ De kat antwoordt "Ki!"
- ✓ De bananen antwoorden "Ki, wie?"
- ✓ De kat antwoordt "Nee, dank je. Ik heb liever bananen!"

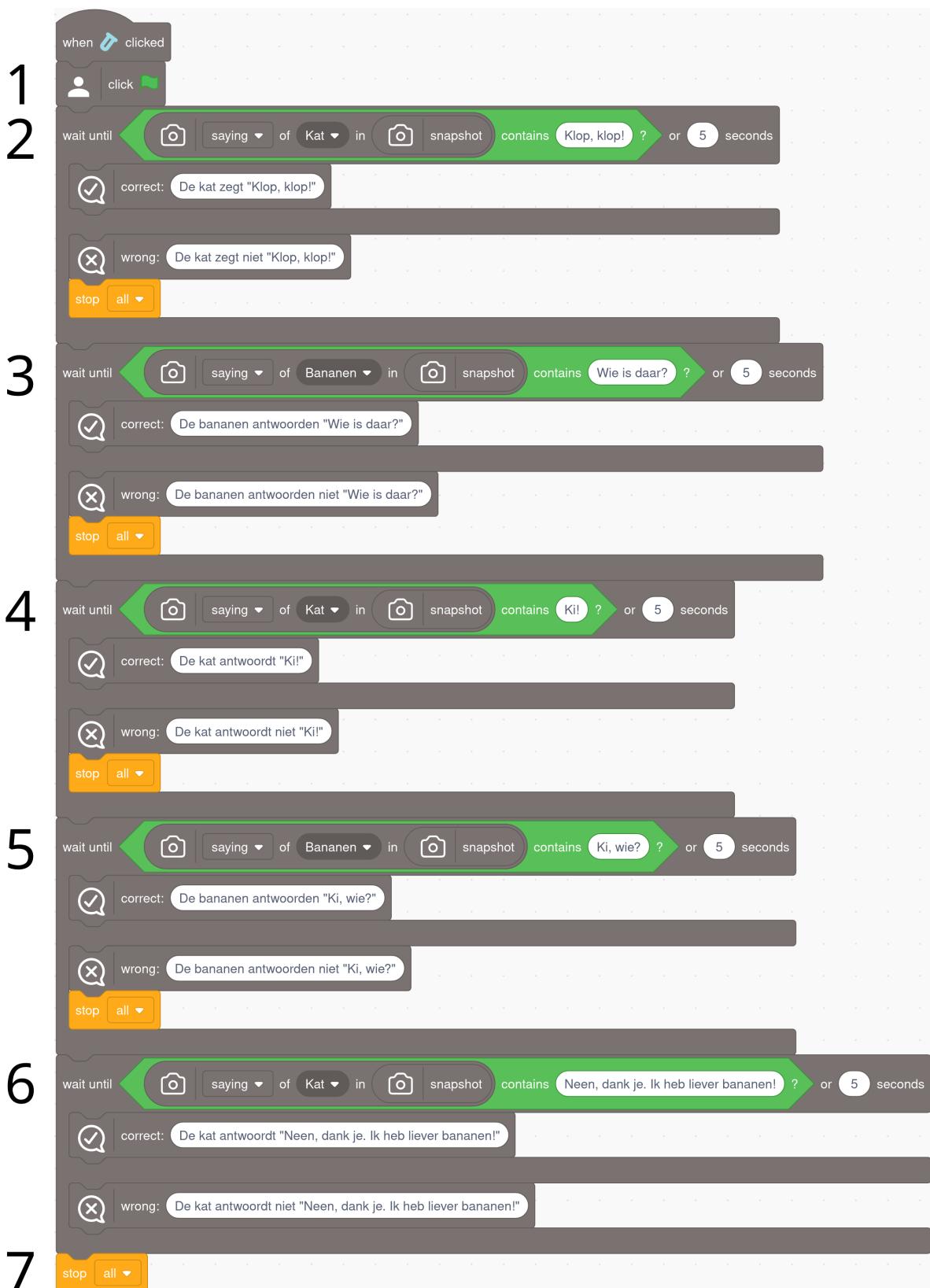
Een voorbeeld van de feedback als een van de sprites iets vergeet te zeggen, of er iets fout gezegd wordt:

- ✓ De kat zegt "Klop, klop!"
- ✓ De bananen antwoorden "Wie is daar?"
- ✓ De kat antwoordt "Ki!"
- ✗ De bananen antwoorden niet "Ki, wie?"

Dit is een voorbeeld van testcode die de uitvoering stopt zodra een controle faalt. Dit wordt gedaan omdat er synchronisatie moet zijn tussen test- en programmacode. Als de tweede controle faalt, en er dus 5 seconden gewacht is geweest, zal de programmacode, waar bijvoorbeeld elke 2 seconden iets gezegd wordt, al een aantal zinnen verder zitten. Controles die na de gefaalde controle gedaan worden, zouden hier dus niet meer betrouwbaar zijn.

Appendix B bevat een completere versie van de testcode waar telkens gewacht wordt tot de sprite eender wat zegt, en dan hetgeen dat gezegd wordt gecontroleerd wordt. Met het simpelere testplan uit Figuur 58 is het bijvoorbeeld mogelijk om 5 keer na elkaar alle zinnen heel snel in omgekeerde volgorde te zeggen en toch door alle testen te raken. Ook kan er duidelijkere feedback gegeven worden door, als er iets fout gezegd wordt, de foute zin in de feedback mee te geven.

Als conclusie stellen we dat deze oefening grotendeels kan getest worden met weinig Scratchcode dankzij de Poke-uitbreiding. Het is echter wel zo dat bepaalde extremere oplossingen geen goeie feedback zullen krijgen.



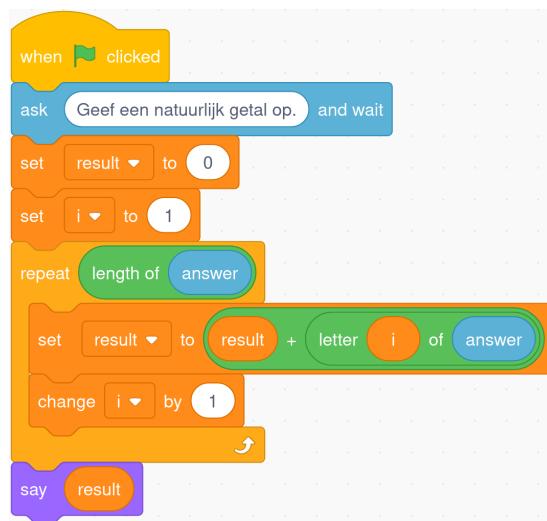
Figuur 58: Testcode voor de oefening “flauw mopje”.

5.3 Cijfersom

Opgave

Het speelveld bevat een kat. Schrijf een programma dat de kat een natuurlijk getal (geen kommagetal) laat vragen. Daarna moet de kat zeggen wat de som van de cijfers van dit getal is. Als de gebruiker bijvoorbeeld het getal 3582 invoert, dan moet de kat het getal 18 ($3 + 5 + 8 + 2$) zeggen. Het programma mag pas beginnen wanneer op de groene vlag wordt geklikt.

In deze oefening zal de sprite “Kat” zaken zeggen en vragen. Hierdoor hoort de oefening thuis in de categorie “praten”. Een modeloplossing is te zien in Figuur 59. Hier vragen we naar een natuurlijk getal. Daarna zetten we de Scratchvariabele “result” op 0. In deze variabele zal het resultaat komen van de berekening. Ook zetten we de Scratchvariabele “i” op 1. Dit zal de index aangeven terwijl we over elk cijfer in “answer” lopen. Dan lopen we over elk cijfer in “answer” en tellen we dit cijfer telkens op bij het resultaat. Uiteindelijk zeggen we het resultaat.



Figuur 59: Een voorbeeldoplossing voor de oefening “cijfersom”. Deze blokken bevinden zich in de sprite “Kat”.

Onze testcode (Figuur 60) doet het volgende:

1. Klik op de groene vlag.
2. Kies een random getal dat we later als antwoord zullen geven.
3. Wacht tot de sprite “Kat” iets vraagt.
 - a) Controleer wat de kat vraagt.

- b) Antwoordt met ons random getal.
4. Bereken zelf de oplossing, gegeven het random getal.
5. Wacht tot de sprite “Kat” iets zegt.
 - a) Controleer dat de kat het correct berekende getal zegt door dit te vergelijken met onze berekende oplossing.
6. Stop programma- en testcode.

De feedback die deze testcode genereert bij de modeloplossing is als volgt:

- ✓ De kat vroeg om een natuurlijk getal.
- ✓ Als er 615 wordt geantwoord zegt de kat 12

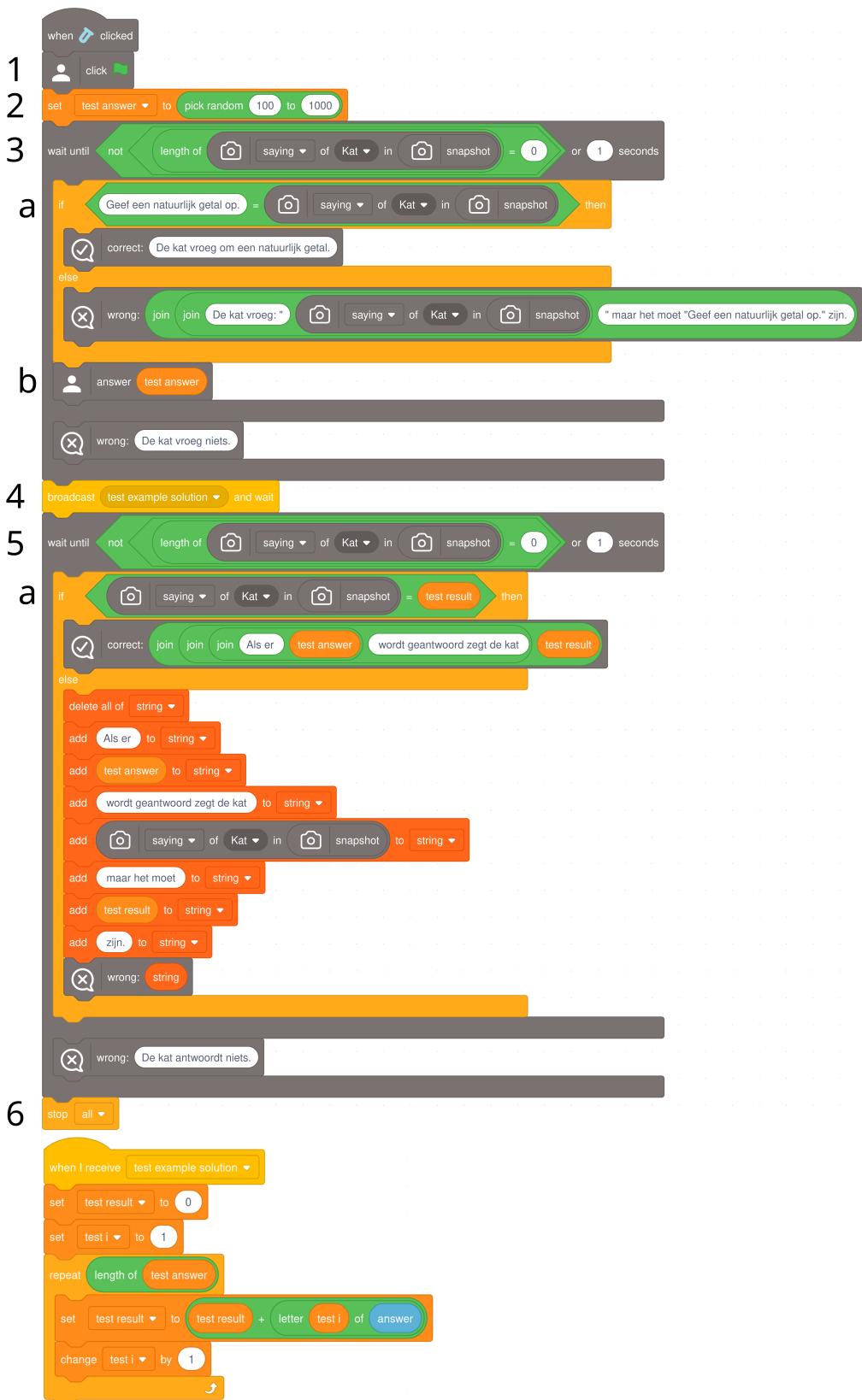
Een voorbeeld van de feedback als de Scratchvariabele “resultaat” per ongeluk op 1 werd geïnitialiseerd in plaats van op 0:

- ✓ De kat vroeg om een natuurlijk getal.
- ✗ Als er 716 wordt geantwoord zegt de kat 15 maar het moet 14 zijn.

In deze testcode zagen we dat de modeloplossing gebruikt werd tijdens het testen. Dit is dus een voorbeeld waar het belangrijk is dat kinderen de testen niet kunnen zien. Door het kiezen van een random getal is de test ook bij elke uitvoer anders waardoor de programmacode niet gewoon een hardgecoedeerd getal kan antwoorden om door de testen te raken.

Een betere test zou zijn om een aantal getallen te testen. Hier willen we dan liefst kunnen controleren dat de programmacode stopte met uitvoeren voor we opnieuw op de groene vlag drukken om zo een aantal keren dezelfde test te kunnen uitvoeren.

Deze oefening is vergelijkbaar met het testen van tekstuele programmeertalen waar er een input wordt gegeven aan de testcode en een output wordt verwacht.

**Figuur 60:** Testcode voor de oefening “cijfersom”.

5.4 Heksenjacht

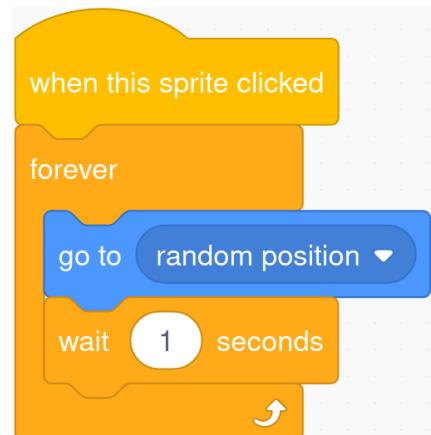
Opgave

Op het speelveld staat een heks. Schrijf een programma dat ervoor zorgt dat de heks op een willekeurige plaats tevoorschijn komt. Het programma mag pas van start gaan wanneer de heks geklikt wordt.

In deze oefening zal de sprite “Heks” zich verplaatsen. Daarom hoort de oefening thuis in de categorie “bewegende sprites”. Het Scratchcanvas dat bij deze oefening hoort, is te zien in Figuur 61a. Een modeloplossing is te zien in Figuur 61b. Deze zal, nadat er op de sprite “Heks” geklikt werd, tot er op de rode “stop”-knop gedrukt wordt telkens de heks op een random positie zetten en 1 seconde wachten.



(a) Het Scratchcanvas



(b) Een voorbeeldoplossing die zich bevindt in de sprite ”Heks”.

Figuur 61: De oefening ”heksenjacht”.

Onze testcode (Figuur 62) doet het volgende:

1. Neem een snapshot.
2. Zet de Scratchvariabele “counter” op 0.
3. Klik op de sprite “Heks”.
4. Reset de timer.

Doe 5 keer:

5. Controleer dat de heks binnen de 1.05 seconde van positie veranderd, als dit niet zo is, geef negatieve feedback en stop de test- en programmacode.

6. Als dit wel zo is, controleer dat de heks niet te snel van positie veranderd. De voorwaarde wordt later beter uitgelegd.
7. Geeft correcte feedback.
8. Stop test- en programmacode.

De voorwaarde bij punt 6 is iets ingewikkelder omdat we hier rekening houden met verschillende interpretaties van de opgave. Het is namelijk niet duidelijk of er, na het klikken op de heks, eerst een seconde moet gewacht worden, of dat het wachten pas na het verplaatsen van de heks moet gebeuren. De blokken in de *forever* lus van Figuur 61b kunnen dus omgewisseld worden. De voorwaarde zal achterhalen welke implementatie wordt gebruikt. Bij de eerste verplaatsing (`counter = 0`):

- Als `timer < 0.05`, zal de oplossing degene zijn te zien in Figuur 61b, want er is bijna geen tijd verstreken. Hier wordt dus eerst de heks verplaatst.
- Als `0.05 < timer < 1`, zal het zo zijn dat er eerst gewacht werd, maar dat de wachttijd niet lang genoeg was. Er moet dus foute feedback gegeven worden.

De feedback die onze testcode genereert bij de voorbeeldoplossing is de volgende:

 De heks is 5 keer van positie veranderd.

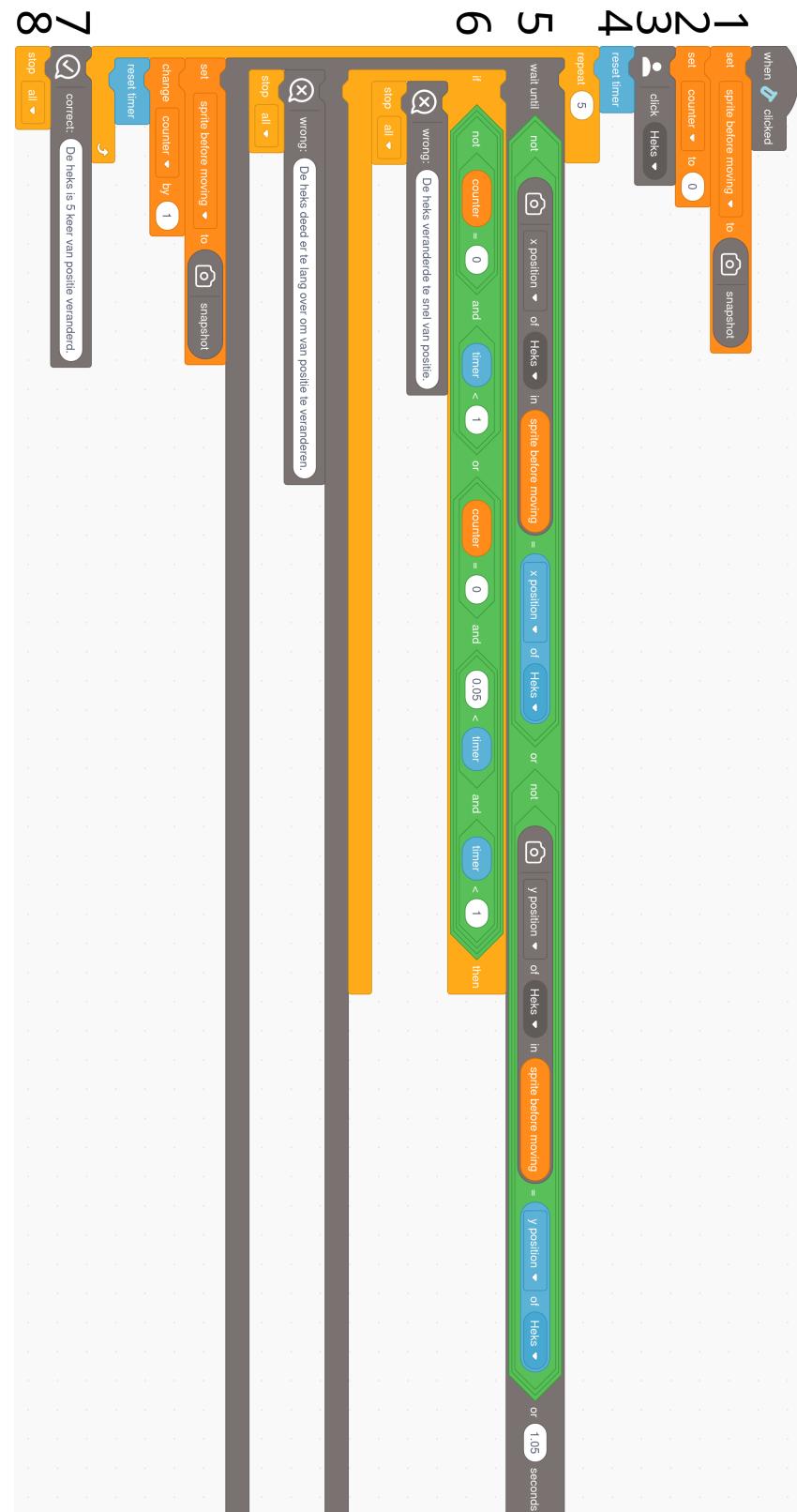
Als de heks niet van plaats verandert binnen 1.05 seconden:

 De heks deed er te lang over om van positie te veranderen.

Als de heks te snel van plaats verandert:

 De heks veranderde te snel van positie.

Het nadeel aan deze testaanpak is dat ze heel soms zal falen als de heks per ongeluk op dezelfde willekeurige positie terechtkomt. Dit is toegestaan volgens de opgave, maar zal niet geaccepteerd worden door de testen. Dit zou moeilijk op te vangen zijn tijdens het testen omdat Poke focus heeft op gedragstesten, en het gedrag van de heks hier juist en fout kan zijn tegelijk.



Figuur 62: Testcode voor de oefening “heksenjacht”.

5.5 Mad hatter

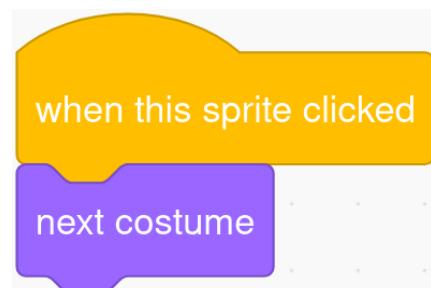
Opgave

Op het speelveld staat een mannetje, Nori, met een gekke hoed op zijn hoofd. Deze hoed is een sprite met vijf uiterlijken. Zorg ervoor dat het volgende uiterlijk van de hoed getoond wordt als je erop klikt.

In deze oefening zal de sprite “Hat” van uiterlijk veranderen. Daarom hoort de oefening thuis in de categorie “animerende sprites”. Het Scratchcanvas dat bij deze oefening hoort, is te zien in Figuur 63a. Een modeloplossing is te zien in Figuur 63b. Hier zullen we simpelweg het volgende uiterlijk tonen als er op de sprite “Hat” geklikt wordt.



(a) Het Scratchcanvas



(b) Een voorbeeldoplossing die zich bevindt in de sprite ”Hat”.

Figuur 63: De oefening ”mad hatter”.

Onze testcode (Figuur 64) zal 10 keer het volgende doen:

1. Neem een snapshot.
2. Klik op de sprite “Hat”
3. Controleer dat het volgende uiterlijk getoond wordt.

De feedback die onze testcode genereert bij de voorbeeldoplossing is de volgende:



Figuur 64: Testcode voor de oefening “mad hatter”.

- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.
- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.
- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.
- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.
- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.
- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.
- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.
- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.
- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.
- ✓ Als er op de hoed geklikt wordt het volgende uiterlijk getoond.

Omdat we hier in een lus feedback gaven met het *assert* blok, zien we dezelfde feedback 10 keer. De testcode gebruikt hier het *click sprite and wait* blok. Als de programmacode (de thread van het *when this sprite clicked* blok) oneindig lang loopt, dat de testcode ook oneindig lang zal vasthangen. Poke voorziet momenteel nog geen mogelijkheid om hiervoor een time-out in te stellen, maar dit is gemakkelijk toe te voegen omdat de implementatie analoog is aan het *wait until or timeout* blok.

5.6 Voetballende kat

Opgave

Op het speelveld staan een kat en een voetbal. Schrijf een programma dat start wanneer je op de spatiebalk drukt, en de kat laat stappen tot aan de voetbal. Wanneer de kat de bal raakt, moet hij stoppen.

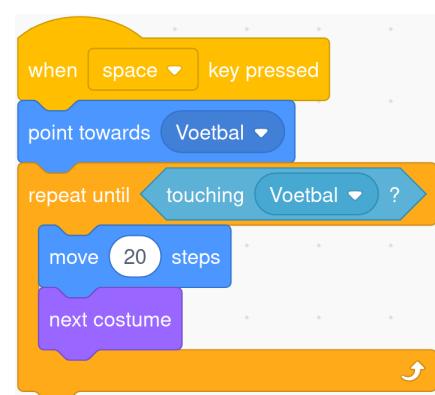
Tip: gebruik het *richt naar* blok.

In deze oefening zal de sprite “Kat” zich verplaatsen naar de voetbal. Ook zal de sprite van uiterlijk veranderen om een stappende animatie te creëren. Daarom hoort de oefening thuis in de zowel de

categorie “bewegende sprites” als de categorie “animerende sprites”. Het Scratchcanvas dat bij deze oefening hoort is te zien in Figuur 65a. De sprite “Kat” heeft twee kostuums die voor een wandelende animatie zorgen als er snel tussen gewisseld wordt. Een modeloplossing is gegeven in Figuur 65b. Deze zal starten als op de spatie-toets gedrukt wordt, waarna de sprite “Kat” naar de voetbal gericht wordt. Dan zal de sprite “Kat” 20 stapjes zetten en van kostuum veranderen tot hij de voetbal aanraakt.



(a) Het Scratchcanvas



(b) Een voorbeeldoplossing die zich bevindt in de sprite ”Kat”.

Figuur 65: De oefening ”voetballende kat”.

Onze testcode (Figuur 66) doet het volgende:

1. Zet de kat voor het doel.
2. Neem een snapshot van de omgeving
3. Druk op de spatie-toets
4. Controleer dat de kat beweegt.
5. Controleer dat de kat een stapje zet door van uiterlijk te veranderen.
6. Controleer dat de kat de voetbal uiteindelijk aanraakt.
7. Stop programma- en testcode.

De testcode zal andere feedback geven bij andere programmacode.

Bij de modeloplossing:

- ✓ De kat beweegt
- ✓ De kat zet een stapje
- ✓ De kat loopt helemaal tot aan de voetbal

Als de kat niet beweegt, maar wel ter plaatste zijn benen beweegt:

- De kat beweegt niet.
- De kat beweegt zijn benen.
- De kat loopt niet tot aan de voetbal.

Als de kat zijn benen niet beweegt, maar wel naar de voetbal loopt:

- De kat beweegt.
- De kat beweegt zijn benen niet.
- De kat loopt helemaal tot aan de voetbal.

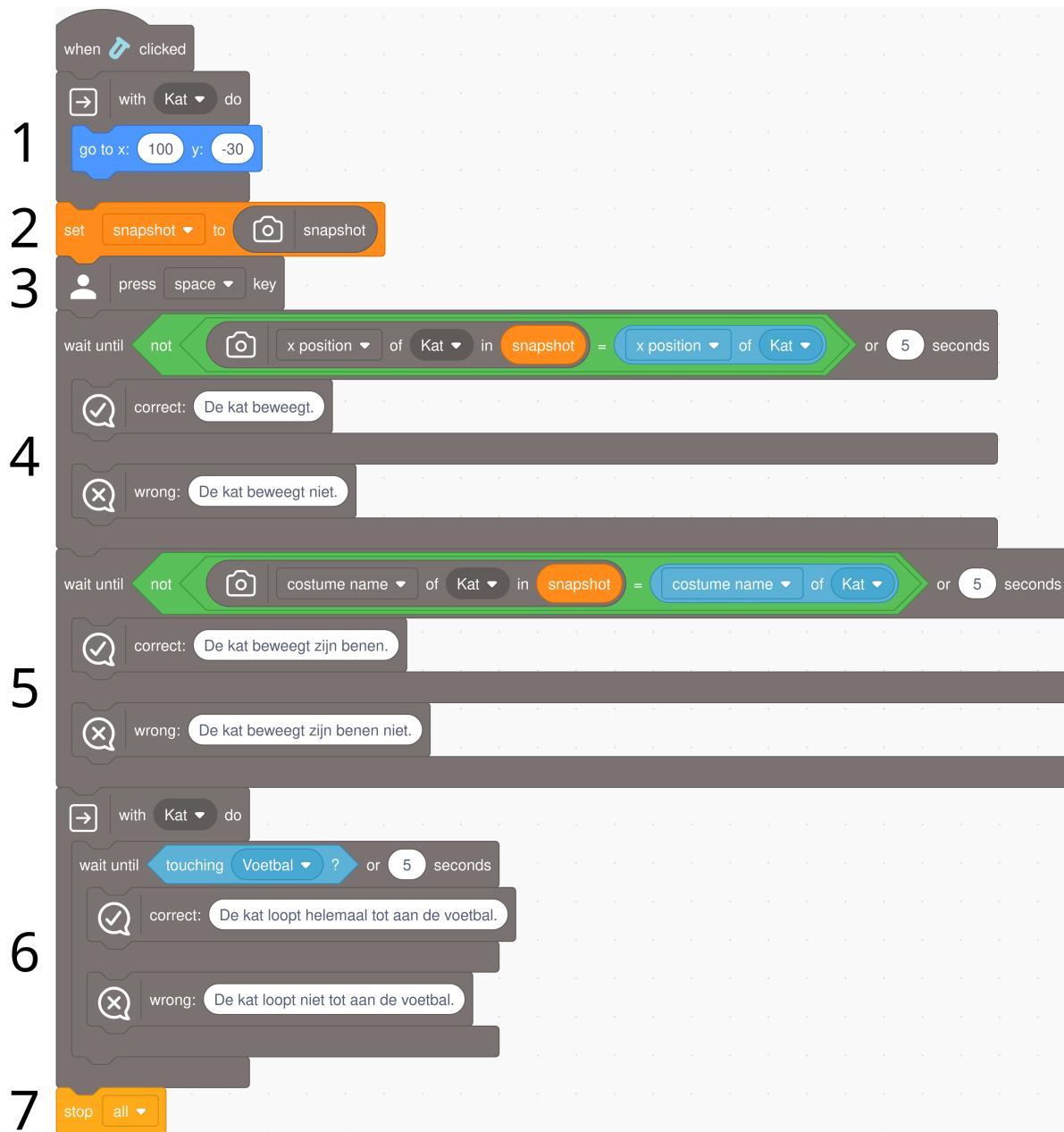
Als de kat naar de verkeerde locatie loopt en zijn benen beweegt:

- De kat beweegt.
- De kat beweegt zijn benen.
- De kat loopt niet tot aan de voetbal.

We kozen deze testcode omdat het gemakkelijk verstaanbaar is en belangrijke eigenschappen van de programmacode test. In de opgave werd ook nog gevraagd dat de kat stopt met lopen als hij bij de bal is. Dit kan ook getest worden door in het 3de *wait until or timeout* blok, in de eerste tak extra controles toe te voegen. Deze controles zouden opnieuw de animatie en de beweging van de kat moeten controleren. Hier deden we dat niet omdat de testcode dan te groot werd.

We zien dus dat er dankzij Poke met weinig blokken testcode kan geschreven worden die duidelijke feedback geeft in verschillende omstandigheden.

Ook hier zou het handig zijn dat Poke de programmacode zou kunnen stoppen. De opgave geeft niet aan dat de kat altijd voor het doel staat. De kat moet naar de voetbal lopen van overal op het speelveld. Poke zou hier de testcode in een lus kunnen zetten en meerdere keren op de spatie toets kunnen drukken waarbij de kat telkens op een andere positie start. Maar omdat er geen manier is om te controleren dat programmacode klaar is met uitvoeren, zouden we de assumptie moeten maken dat de programmacode vanzelf stopte nadat de kat de voetbal aanraakte.



Figuur 66: Testcode voor de oefening “voetballende kat”.

5.7 Vliegende papegaai

Opgave

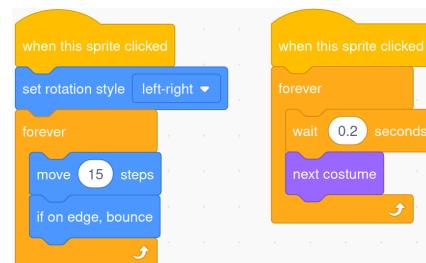
Op het speelveld staat een papegaai: een sprite met twee uiterlijken. Zorg ervoor dat de papegaai van links naar rechts begint te vliegen van zodra hij wordt aangeklikt. Tijdens het vliegen moet de papegaai met zijn vleugels klapperen. Wanneer de papegaai de rand raakt, moet hij omkeren.

Tip: gebruik voor het vliegen het *verander uiterlijk naar* blok

In deze oefening zal de sprite “Papegaai” heen en weer vliegen. Ook zal de sprite van uiterlijk veranderen om een vliegende animatie te creëren. Daarom hoort de oefening thuis in de zowel de categorie “bewegende sprites” als de categorie “animerende sprites”. Het Scratchcanvas dat bij deze oefening hoort, is te zien in Figuur 67a. Een modeloplossing is te zien in Figuur 67b. We zien hier 2 startblokken die een thread zullen starten wanneer op de sprite “Papegaai” geklikt wordt. De stapel blokken die bij het eerste startblok hoort zal eerst de draaistijl op “links-rechts” zetten. Dit beslist hoe de papegaai moet draaien als hij tegen de rand vliegt. Daarna zal in een oneindige lus 15 stappen gezet worden, en als de papegaai de rand raakt zal deze omdraaien. De tweede stapel blokken zorgt voor de animatie van de papegaai en zal het uiterlijk elke 0.2 seconden wisselen.



(a) Het Scratchcanvas



(b) Een voorbeeldoplossing die zich bevindt in de sprite ”Hat”.

Figuur 67: De oefening ”vliegende papegaai”.

Onze testcode (Figuur 68) doet het volgende:

1. Zet de papegaai klaar zodanig dat hij naar de linkse muur kijkt.
2. Neem een snapshot.
3. Klik op de papegaai.

4. Controleer dat de papegaai beweegt.
 - a) Als de papegaai beweegt, controleer dat hij zich omdraait als hij tegen de rand botst.
5. Controleer dat de papegaai zijn vleugels flappert door van uiterlijk te veranderen.
6. Stop de test- en programmacode.

De voorwaarde bij 4a gebruikt de x-positie om te controleren dat de papegaai een stuk terug is gevlogen en zijn richting heeft veranderd.

De feedback die onze testcode genereert bij de voorbeeldoplossing is de volgende:

- ✓ De papegaai beweegt.
- ✓ De papegaai draait zich om als hij tegen de rand vliegt.
- ✓ De papegaai fladdert zijn vleugels.

Als de papegaai niet van uiterlijk verandert:

- ✓ De papegaai beweegt.
- ✓ De papegaai draait zich om als hij tegen de rand vliegt.
- ✗ De papegaai fladdert zijn vleugels niet!

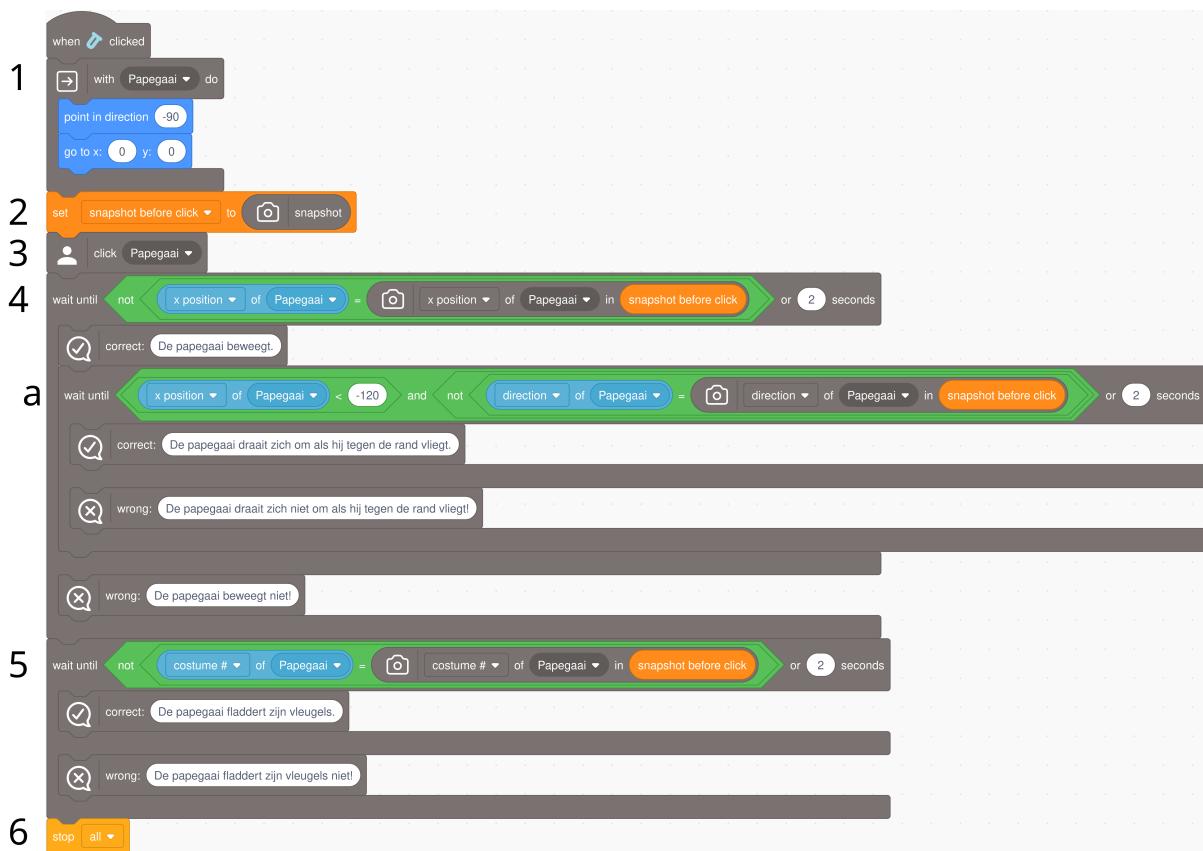
Als de papegaai niet beweegt:

- ✗ De papegaai beweegt niet!
- ✓ De papegaai fladdert zijn vleugels.

Als de papegaai wel beweegt, maar zich niet omdraait als hij tegen de rand botst.

- ✓ De papegaai beweegt.
- ✗ De papegaai draait zich niet om als hij tegen de rand vliegt!
- ✓ De papegaai fladdert zijn vleugels.

In deze oefening zou het gebruik van een logboek zoals in Itch gedaan wordt waarschijnlijk het testen een stuk gemakkelijker maken. Dit omdat we de papegaai dan gewoon kunnen laten vliegen, om dan achteraf te controleren dat de papegaai bewoog, zijn vleugels fladderde en zich omkeerde bij het botsen tegen de rand.



Figuur 68: Testcode voor de oefening “vliegende papegaai”.

5.8 Vang de appels

Opgave

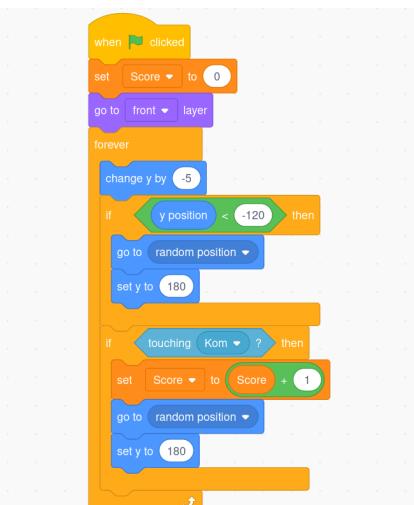
Het speelveld bevat een kom en een appel. De appel begint naar beneden te vallen van zodra op de groene vlag gedrukt wordt. De kom beweegt naar links en naar rechts als je met de muis beweegt.

Pas het programma aan, zodat er linksboven een scoreveld komt te staan, dat bijhoudt hoeveel appels er zijn opgevangen in de kom. Als op de groene vlag gedrukt wordt, dan moet de score op nul gezet worden. Tip: Gebruik een variabele.

In deze oefening zal de leerling de muis gebruiken om de sprite “Kom” te bewegen en appels te vangen. De muisaanwijzer zal dus nodig zijn om constante gebruikersinteractie te simuleren. Daarom hoort de oefening thuis in de categorie “spelletjes”. Het Scratchcanvas dat bij deze oefening hoort, is te zien in Figuur 69a. Een modeloplossing is te zien in Figuur 69b. Het meeste van deze oplossing wordt al gegeven aan de leerling, alleen de scoretelling moet nog toegevoegd worden. Na het indrukken van de groene vlag zal de Scratchvariabele “Score” op 0 gezet worden. Daarna hebben we de code die de appel doet vallen. De lus bevat 2 if-statements. De eerste zal de appel terug naar boven verplaatsen als de appel te laag valt. De tweede zal de appel naar boven verplaatsen als deze de sprite “Kom” aanraakt, en zal ook de score verhogen. De sprite “Kom” bevat code om de muisaanwijzer te volgen, maar deze is voor ons niet relevant.



(a) Het Scratchcanvas



(b) Een voorbeeldoplossing die zich bevindt in de sprite “Appel”.

Figuur 69: De oefening “vang de appels”.

Onze testcode (Figuur 71) doet het volgende:

1. Zet de appel bovenaan het speelveld.
2. Zet de Scratchvariabele “test score” op 0.
3. Start de gebruikerssimulatie om de sprite “Kom” te bewegen.
4. Klik op de groene vlag.
5. Maak een testgroep.

Doe 5 keer:

6. Wacht tot de y-positie van de appel een grote sprong maakt, dit betekent dat de appel net gevangen werd en de sprite terug naar boven werd gezet.
7. Tel 1 op bij de Scratchvariabele “test score”.
8. Controleer dat de Scratchvariabele “Score” hetzelfde is als de Scratchvariabele “test score”.
9. Stop de test- en programmacode.

Om de waarde van de Scratchvariabele “Score” te weten te komen, maken we gebruik van het *get* blok. Dit blok werd niet uitgelegd in Hoofdstuk 2 en werd later toegevoegd toen bleek dat het nuttig kan zijn om de waarde van een Scratchvariabele uit een snapshot te kunnen halen.

Dit is de feedback die onze testcode genereert bij de voorbeeldoplossing:

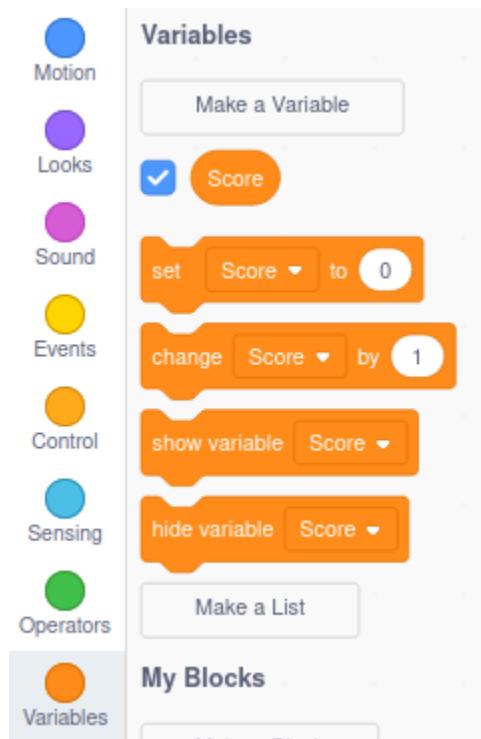
- ▼ ✓ Vang 5 appels
- ✓ Na 1 appel(s) vangen was de score juist
 - ✓ Na 2 appel(s) vangen was de score juist
 - ✓ Na 3 appel(s) vangen was de score juist
 - ✓ Na 4 appel(s) vangen was de score juist
 - ✓ Na 5 appel(s) vangen was de score juist

Als de Scratchvariabele “Score” op 1 wordt geïnitialiseerd in plaats van 0 zullen alle scores 1 te hoog zijn:

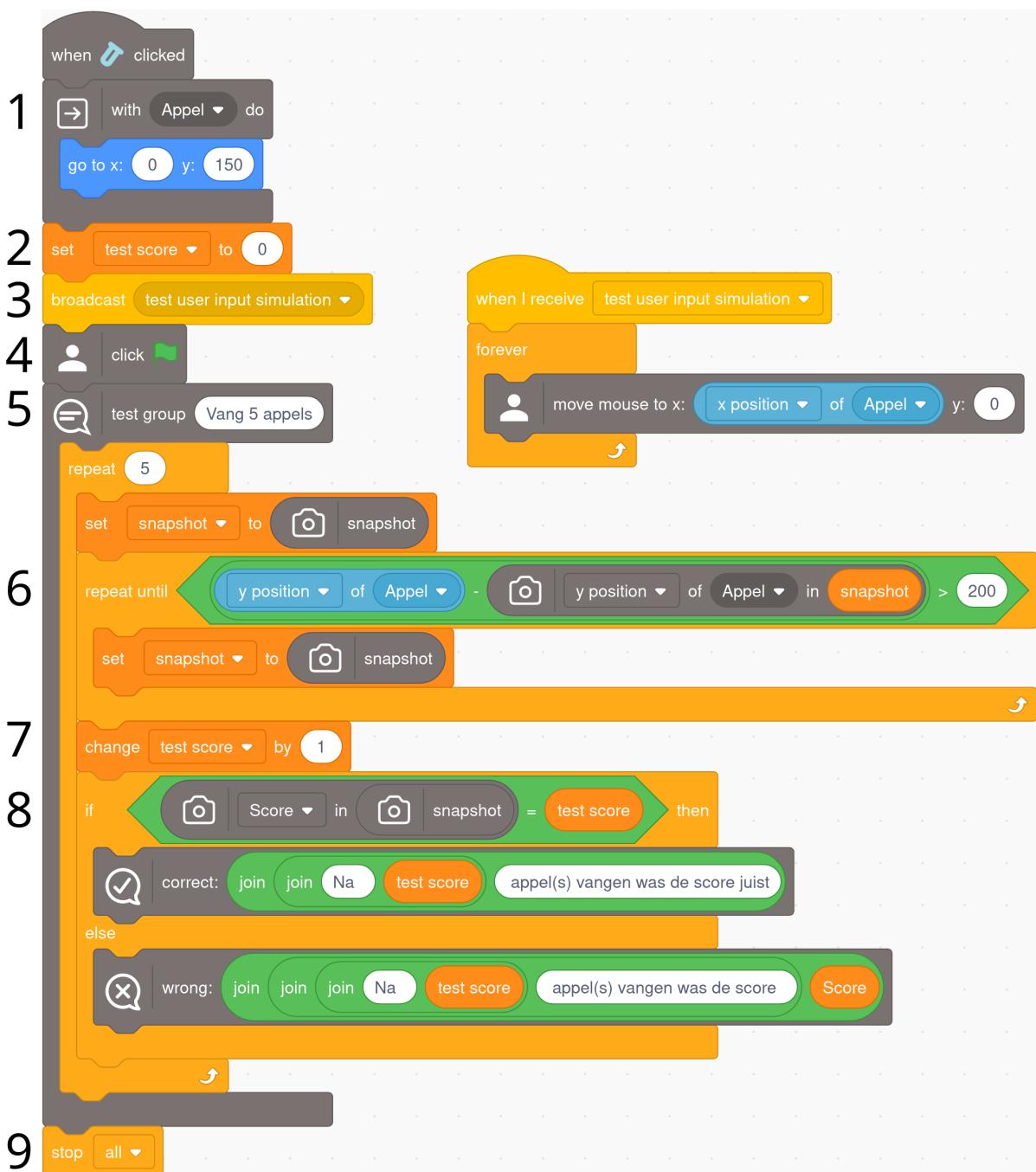
- ▼ ✗ Vang 5 appels
- ✗ Na 1 appel(s) vangen was de score 2
 - ✗ Na 2 appel(s) vangen was de score 3
 - ✗ Na 3 appel(s) vangen was de score 4
 - ✗ Na 4 appel(s) vangen was de score 5
 - ✗ Na 5 appel(s) vangen was de score 6

De opgave geeft aan dat de variabele moet getoond worden zoals te zien is in Figuur 69a. Dit moet door de variabele aan te vinken in het blokkenmenu (Figuur 70). Of een variabele getoond wordt of

niet kan momenteel niet gecontroleerd worden met Poke. Ook is Poke gelimiteerd in het opvragen van variabelen die tijdens het opstellen van de testen nog niet bestaan. Variabelen die leerlingen zelf moeten aanmaken als deel van de opgave zullen namelijk nog niet beschikbaar zijn in het dropdown menu van het *get* blok tijdens het schrijven van de test. Om deze opgave te kunnen testen zal de variabele "Score" dus al moeten bestaan in het Scratchproject die de leerling krijgt.



Figuur 70: De Scratchvariabele "Score" is aangevinkt zodat deze getoond wordt.



Figuur 71: Testcode voor de oefening “vang de appels”.

5.9 Tekenen

Er zijn 3 oefeningen die gebruikmaken van de Penuitbreiding om figuren te tekenen:

- Teken een vierkant
- Teken een driehoek
- Teken een huis

Poke ondersteunt het gebruik van de Penuitbreiding nog niet, en zoals besproken in de Itchmaster-proef is het tekenen van figuren vaak te subjectief om goed te testen.

5.10 Conclusie en toekomstige uitbreidingen

In dit hoofdstuk bekeken we Poke testen voor oefeningen die gebruikt werden in de jeugdrondes van de Vlaamse Programmeerwedstrijd in 2017. We deelden deze oefeningen op in verschillende categorieën:

	Testbaar met Poke
Praten	Grotendeels. Poke kan controleren dat sprites bepaalde zaken zeggen/denken en kan ook gestelde vragen beantwoorden. Poke kan momenteel nog geen onderscheid maken tussen tekst die gevraagd of gezegd wordt door een sprite.
Bewegende sprites	Ja. Poke kan de locatie van sprites controleren.
Animerende sprites	Ja. Poke kan controleren welk uiterlijk van een sprite getoond wordt.
Spelletjes	Ja. Poke kan constante gebruikersinteractie simuleren en de omgeving opvragen.
Tekenen	Nee. De Penuitbreiding is momenteel nog niet ondersteund.

We zien dus dat Poke nuttig kan gebruikt worden voor heel wat oefeningen. Uit deze oefeningen leren we ook dat Poke minstens nog de volgende functionaliteit mist:

- Een blok dat programmacode, en alleen programmacode, kan stoppen, en een manier om te controleren dat programmacode vanzelf gestopt is.

- Een manier om te controleren dat een variabele getoond wordt op het canvas.
- Er wordt al onderscheid gemaakt tussen iets dat gedacht wordt en iets dat gezegd wordt, maar nog niet tussen iets dat gezegd wordt en iets dat gevraagd wordt.
- Een time-out bij de *and wait* blokken en mogelijks ook het *answer* blok.
- Statische testen van codeblokken en variabelen.
- Automatisch de gebruikersinteractie toevoegen aan de feedbackboom zodat een leerling een overzicht heeft van welke gebruikersinteractie zich voordeed tijdens het testen.
- Het gebruik van een logboek zoals in Itch mogelijk maken.

6 Conclusie

In deze masterproef onderzochten we of een testframework voor Scratch kon ontwikkeld worden waarmee testcode met Scratchblokken kan geschreven worden. Ook bekeken we welke Scratchblokken nodig waren om zo ergonomisch mogelijke testcode te schrijven. We maakten een proof-of-concept implementatie van een blokgebaseerd testframework. De implementatie gebeurde via een Scratchuitbreiding genaamd Poke. We probeerden zoveel mogelijk aan te leunen bij de filosofie van Scratch. Dit is bijvoorbeeld zichtbaar bij de *and wait* blokken waar de inspiratie kwam van het bestaande *broadcast and wait* blok. De UBAS-blokken zorgde voor een ergonomischere testervaring omdat het met deze blokken mogelijk is de testcode overzichtelijk in 1 sprite te schrijven.

Poke laat ons toe testcode te schrijven en uit te voeren om Scratchprojecten te beoordelen. Testcode kan automatisch feedback geven aan kinderen tijdens het programmeren door het gebruik van de feedbackboom.

We bekommerden ons niet over hoe het testen final in de praktijk gebracht wordt om meerdere oplossingen te testen. Ook was het genereren van zo duidelijk mogelijke feedback geen hoofddoel.

Uiteindelijk sloten we deze masterproef af met een reeks oefeningen die als validatie diende voor de Poke-uitbreiding. Daar zagen we dat Poke voor een reeks oefeningen nuttig kan gebruikt worden.

6.1 Toekomstig werk

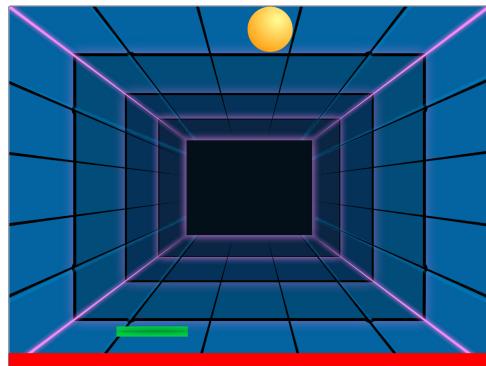
In Paragraaf 5.10 bespraken we al een aantal missende functionaliteiten, gebaseerd op de gebreken dat we opmerkten tijdens het testen van de oefeningen. Nu bekijken we nog een aantal werkpunten van Poke:

- Poke kan nog niet overweg met klonen. Een blok analoog aan het *sprite filter* blok dat tijdens uitvoering klonen kan selecteren zou waarschijnlijk handig zijn tijdens het testen van oefeningen met klonen.
- De feedbackboom werkt, maar is een ietwat primitieve vorm van feedback. Een eerste verbetering kan zijn om feedback die meerdere keren hetzelfde zegt (zoals bij de oefening “mad hatter”, Paragraaf 64) te aggregeren. Uiteindelijk lijkt het ons ideaal om een integratie met de debugger (De Proft e.a. 2022) te hebben, zodat een leerling naar het exacte punt in de tijd kan springen waar iets fout liep tijdens het testen. Ook zouden de gebruikersinteractiemomenten zichtbaar kunnen zijn in de tijdsbalk van de debugger
- De simulatie van de muisaanwijzer wordt momenteel overschreven als de echte muisaanwijzer bewogen wordt. Dit zorgt in de praktijk voor een gevecht tussen de gesimuleerde muisaanwijzer en de echte muisaanwijzer. De input van de echte muisaanwijzer zou moeten geblokkeerd worden tijdens het testen.

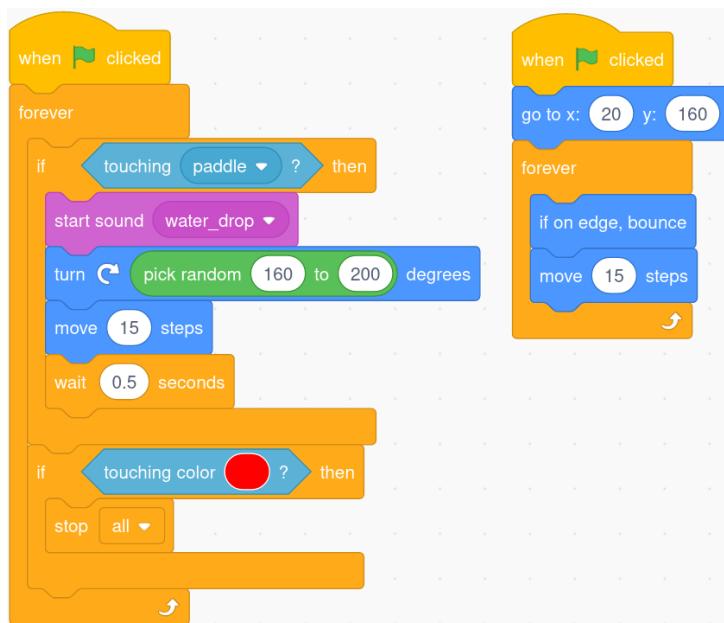
- In Paragraaf 2.7 besproken we al dat Poke niet overweg kan met meerdere *when tests started* blokken. Als in Scratch meerdere dezelfde startblokken gebruikt worden zullen deze allemaal samen activeren en zullen hun threads parallel uitvoeren. Gelijkaardig is het misschien een goed idee om zo meerdere testcodethreads in parallel uit te voeren. Per testcodethread zou dan een instantie van de Scratch VM moeten bestaan waar telkens maar 1 testcodethread in uitvoert. Dit om interferentie tussen verschillende parallel lopende testen te vermijden.

- Andreas, Stahlbauer, Frädrich Christoph, en Fraser Gordon. 2020. ‘Verified from Scratch: Program Analysis for Learners’ Programs’. In ASE. IEEE.
- De Proft, Robbe, Peter promotor Dawyndt, Christophe promotor Scholliers, en Niko [FB] Strijbol. 2022. ‘Blink: een educatieve software-debugger voor Scratch 3.0’. 2022. <http://lib.ugent.be/catalog/rug01:003059967>.
- Ecma, ECMA. 1999. ‘262: Ecmascript language specification’. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*.
- Götz, Katharina, Patric Feldmeier, en Gordon Fraser. 2022. ‘Model-based Testing of Scratch Programs’. arXiv. <https://doi.org/10.48550/ARXIV.2202.06271>.
- Harvey, Brian, Daniel D Garcia, Tiffany Barnes, Nathaniel Titterton, Daniel Armendariz, Luke Segars, Eugene Lemon, Sean Morris, en Josh Paley. 2013. ‘Snap!(build your own blocks)’. In *Proceeding of the 44th ACM technical symposium on Computer science education*, 759–59.
- Mak, Nils, Peter promotor Dawyndt, en Christophe copromotor Scholliers. 2019. ‘Itch: een educatief testframework voor automatische feedback op Scratch projecten’. 2019. <http://lib.ugent.be/catalog/rug01:002787413>.
- Moreno-León, Jesús, en Gregorio Robles. 2015. ‘Dr. Scratch: A Web Tool to Automatically Evaluate Scratch Projects’. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, 132–33. WiPSCE ’15. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2818314.2818338>.
- Resnick, Mitchel, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, e.a. 2009. ‘Scratch: programming for all’. *Communications of the ACM* 52 (11): 60–67.
- Stahlbauer, Andreas, Marvin Kreis, en Gordon Fraser. 2019. ‘Testing Scratch Programs Automatically’. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 165–75. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/3338906.3338910>.
- Wang, Wengran, Chenhao Zhang, Andreas Stahlbauer, Gordon Fraser, en Thomas Price. 2021a. ‘SnapCheck: Automated Testing for Snap Programs’. In *ITiCSE’21 - Proceedings of the 2021 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE’21. Association for Computing Machinery.
- . 2021b. ‘SnapCheck: Automated Testing for Snap! Programs’. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, 227–33. ITiCSE ’21. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/3430665.3456367>.

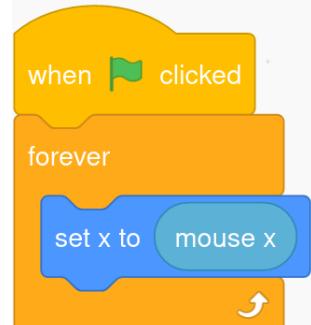
Appendix A



Figuur 72: Het canvas voor de oefening “Pong”.



(a) De programmacode in de sprite ”ball”.

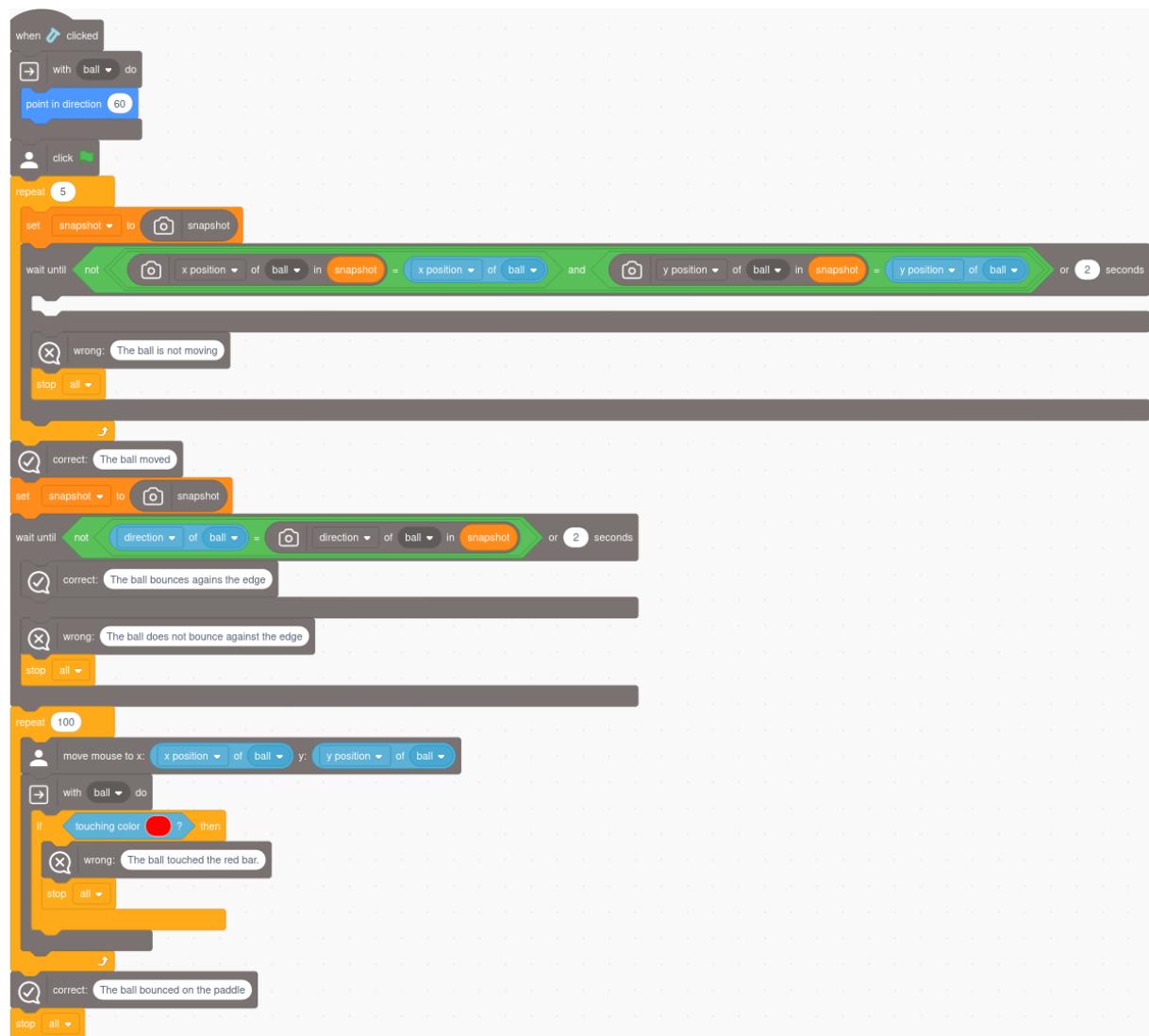


(b) De programmacode in de sprite ”paddle”.

Figuur 73: De programmacode van de oefening ”pong”.

- ✓ The ball moved
- ✓ The ball bounces agains the edge
- ✓ The ball bounced on the paddle

Figuur 74: Feedback voor de oefening “pong”.



Figuur 75: De testcode voor de oefening “pong”.

Appendix B

Een grondigere test voor de oefening “flauw mopje”:

