FACULTY
OF SCIENCES

EDUCATIONAL SOFTWARE TESTING FOR TEXTUAL AND BLOCK-BASED PROGRAMMING LANGUAGES

Niko Strijbol

Supervisors:
Prof. Dr. Peter Dawyndt
Prof. Dr. Christophe Scholliers
Prof. Dr. Ir. Bart Mesuere

A dissertation submitted to Ghent University in partial fulfilment of the requirements for the degree of Doctor of Computer Science.

Academic year: 2023 – 2024

GHENT
UNIVERSITY

This book was typeset using LaTeX and LuaTeX with Markus Kohm's document class KOMA-Script.

The text is set in Frank Grießhammer's Source Serif, an open-source font from Adobe. Titles, headings, and other accents use Paul Hunt's Source Sans. Both were designed to work together, under the auspices of Robert Slimbach. Code is set with Paul Hunt's Source Code Pro, from the same family.

The title page is set in UGent Panno Text by Pieter van Rosmalen, as required by our University's style guide. Originally developed for South Korean traffic signs, the Text variant was optimized for continuous text. Fun fact: the commercially available Panno Text does include italic variants.

Cover illustration by Дарья Гаенко [Darya Gaenko], under licence from iStock. It depicts the DVK 2, a Soviet computer from around 1985.

Samenvatting

Leren programmeren is uitdagend en aldus ervaren veel studenten programmeervakken als moeilijk. Programmeeronderwijs is geen uitzondering op het spreekwoord oefening baart kunst. Het is algemeen aanvaard dat het doen de beste manier is om te leren programmeren: hoe meer ervaring, hoe beter de programmeerkunst. Om evenwel iets te leren van al die programmeerervaring is het belangrijk dat studenten op tijd voldoende kwalitatieve feedback krijgen.

Jammer genoeg is net het geven van die feedback heel tijdrovend en arbeidsintensief, zeker als er veel oefeningen en grote aantallen studenten zijn. Enerzijds moeten studenten dus zoveel mogelijk programmeren, maar anderzijds is er weinig tijd om goede feedback te voorzien. Daarom is er een lange en rijke geschiedenis (sinds de jaren 1960) van het gebruik der automatisering om feedback te geven. Het proces om feedback op geautomatiseerde wijze te geven heet geautomatiseerde beoordeling (van het Engelse *automated assessment*).

Lang is relatief natuurlijk, maar wel gepast wetende dat leren programmeren zelf in de jaren 1960 opkwam.

In de meeste gevallen houdt geautomatiseerde beoordeling voor programmeeronderwijs in dat men werkt met testraamwerken voor software. De code die studenten voor een bepaalde oefeningen indienen (we noemen dit een oplossing) wordt getest, en dat minstens op juistheid. Vaak is de feedback wel veel uitgebreider dan enkel een globale juist of fout.

Onze vakgroep heeft, zoals zovele anderen, een online platform gemaakt voor geautomatiseerde beoordelingen: Dodona. Een belangrijke eigenschap is de scheiding tussen het platform zelf (verantwoordelijk voor gebruikersbeheer, cursusbeheer, de gebruikersinterface, enz.) en de *judge* (het testraamwerk verantwoordelijk voor het beoordelen van oplossingen). Zo kan Dodona bijna elke programmeertaal ondersteunen: momenteel is er ondersteuning voor C, Haskell, Java, Kotlin, Prolog, R, Scheme, Bash, C#, JavaScript, Python, HTML, SQL, Markdown, en Turtle.

Tijdens het werken aan en met Dodona stelden we enige tekortkomingen vast in bestaande hulpmiddelen die gebruikt worden in het programmeeronderwijs. Hoofdstuk 1 geeft een gedetailleerd overzicht van de

onderwijscontext en van het Dodona-platform. Samengevat behandelt dit proefschrift vijf van die waargenomen tekortkomingen.

We merkten dat veel oefeningen in Dodona geschikt zijn om te gebruiken in meerdere programmeertalen, althans in theorie. Om een oefening daadwerkelijk te gebruiken in een andere programmeertaal, moet men ze eerst kopiëren, dan handmatig het testplan omzetten naar het formaat dat de judge voor die programmeertaal gebruikt, en ten slotte nog de configuratiebestanden en opgave aanpassen. Dit is veel handwerk. Hoofdstuk 2 biedt een oplossing: **TESTed**, een educatief testraamwerk voor software. Kenmerkend aan TESTed is de mogelijkheid om programmeertaalonafhankelijke oefeningen te schrijven. Dit wil zeggen dat één oefening (met één enkel testplan) opgelost kan worden in meerdere programmeertalen, met ondersteuning voor geautomatiseerde beoordeling. Een oefeningen is dus bruikbaar in meerdere programmeertalen zonder enige bijkomende inspanning.

Met een prototype van TESTed in de hand namen we dan een stapje terug om naar het grote geheel te kijken: wat is er nodig om van een prototype naar een goede oplossing voor het maken van programmeeroefeningen te gaan? We willen TESTed de standaardoptie maken voor lesgevers, in zowel hoger als secundair onderwijs. Hiervoor hebben we **TESTed-DSL** in het leven geroepen, dat we voorstellen in hoofdstuk 3. Het is een domeinspecifieke taal om oefeningen met ondersteuning voor geautomatiseerde beoordeling in meerdere programmeertalen te schrijven. Een domeinspecifieke taal is een formaat dat specifiek ontworpen is voor een bepaald gebruik, hier dus het schrijven van programmeeroefeningen. Door aandacht te besteden aan de ergonomie van TESTed-DSL, hebben we ervoor gezorgd dat de taal ook nuttig is voor oefeningen die niet bedoeld zijn om gebruikt te worden in meerdere programmeertalen. We raden nu alle lesgevers op Dodona aan om TESTed te gebruiken voor het opstellen van oefeningen, zelfs als ze oefeningen willen maken die bijvoorbeeld enkel in JavaScript moeten opgelost worden.

Bij jonge kinderen gebruikt men vaak visuele programmeertalen om te leren programmeren. Een visuele programmeertaal laat gebruikers toe om programma's te maken door stukken van het programma niet tekstueel maar grafisch te manipuleren. Scratch is binnen het onderwijs veruit de meestgebruikte visuele programmeertaal. Programmeren in Scratch bestaat uit het slepen en in elkaar klikken van blokjes (een beetje zoals puzzelstukjes of legoblokjes). Vandaar dat men Scratch ook wel een blokgebaseerde programmeertaal noemt. Een gedetailleerde inleiding over Scratch staat in hoofdstuk 4.

Dodona ondersteunt meerdere programmeertalen, dus oorspronkelijk wilden we ondersteuning voor Scratch toevoegen aan Dodona. Echter,

Scratch is niet alleen een programmeertaal, het is ook een programmeeromgeving. Het werd snel duidelijk dat een platform voor Scratch andere vereisten heeft dan wat we met Dodona kunnen doen. Daarom gingen we een samenwerking aan met CodeCosmos, een commerciële partner. Aangezien CodeCosmos een educatieve uitgeverij is, die ook oefeningen voor Scratch aanbiedt, heeft ze al een platform voor Scratch. Bovendien heeft ze ook meer ervaring met het maken van oefeningen voor Scratch.

Hoofdstuk 5 stelt **Itch** voor, ons testraamwerk voor Scratch. Het ondersteunt zowel statische testen (wat betekent dat er enkel naar de blokken gekeken wordt, zonder het programma uit te voeren) en dynamische testen (waar het programma uitgevoerd wordt met een bepaalde invoer en de resultaten bekeken worden). Deze combinatie betekent dat Itch een diverse reeks Scratch-programma's kan beoordelen. Scratch lijkt in bepaalde opzichten meer op een spelletje dan op een programmeertaal. Als gevolg hiervan experimenteren kinderen veel en gebruiken ze hun fantasie bij het programmeren. Dat is op zijn beurt een uitdaging bij het testen van Scratch-programma's. Als de opgave bijvoorbeeld "Teken een huis" is, hoe kunnen we een oplossing hiervoor dan beoordelen? Er zijn dus limieten aan de soorten oefeningen die Itch kan beoordelen. De overwegingen die bij deze beslissingen komen kijken worden behandeld in het hoofdstuk.

Als een testraamwerk zoals Itch feedback geeft aan leerlingen, dan is alles soms juist, maar veel vaker zijn er testen die falen. Daarop begint het debugproces: leerlingen moeten achterhalen wat de oorzaak van de gefaalde test is. Dit is notoir moeilijk, want de locatie van de oorzaak in het programma is vaak niet voor de hand liggend. Er zijn gelukkig wel middelen om hiermee te helpen, met als belangrijkste de debuggers. Voor tekstuele programmeertalen zijn er veel debuggers en is er ook veel onderzoek over debuggers. Dodona ondersteunt bijvoorbeeld een debugger voor Python.

Voor Scratch, en blokgebaseerde programmeertalen in het algemeen, is dit evenwel niet het geval. Daarom introduceren we in hoofdstuk 6 een nieuwe debugger voor Scratch: **Blink**. Onze debugger ondersteunt stappen door de code (stapsgewijs de code uitvoeren), de programma-uitvoering pauzeren en verder laten lopen, breekpunten (speciale blokken die de programma-uitvoering pauzeren wanneer ze zelf uitgevoerd worden), en tijdreizen. Een debugger met tijdreizen geeft de mogelijkheid om terug te spoelen in de uitvoering van het programma. Elke stap in de uitvoering wordt opgeslagen, dus kunnen we nadien stap per stap teruggaan. Omdat Scratch voornamelijk gebruikt wordt door een jong publiek, hebben we veel aandacht besteed aan het intuïtief maken van

de debugger. De eerste experimenten in een klas tonen dat leerlingen inderdaad vinden dat de debugger makkelijk om mee te werken is, en dat ze het tijdreizen in het bijzonder nuttig vinden.

We hebben net gezegd dat de debugger het mogelijk maakt om stapsgewijs een programma uit te voeren. We hebben bewust niet beschreven wat we bedoelen met een stap in de context van Scratch. In Scratch bestaat een project namelijk uit verschillende sprites (die getekend worden op het scherm). Elke sprite heeft zijn eigen code, een verzameling stapels (een stapel is een reeks aan elkaar vastgemaakte blokken). Elke stapel van elke sprite wordt gelijktijdig uitgevoerd in Scratch. Een traditionele definitie van een stap (één blok in één stapel per keer) vinden we daarom niet ideaal. In plaats daarvan willen we bij een stap in elke stapel één blok verder gaan.

Dit is evenwel niet mogelijk door de manier waarop Scratch intern werkt (het uitvoeringsmodel). Scratch gebruikt een coöperatief systeem, wat betekent dat het meerdere blokken in dezelfde stapel uitvoert, dan overschakelt naar de volgende stapel en daar meerdere blokken uitvoert, enzovoort. Door snel tussen stapels te wisselen, lijkt het alsof de stapels in parallel uitgevoerd worden. Dit uitvoeringsmodel werd gekozen om een aantal synchronisatieproblemen bij gelijktijdige programma's te vermijden, maar heeft ook nadelen. Zo veroorzaakt het in bepaalde gevallen niet-intuïtief gedrag.

Snel wisselen om parallelisme na te boosten is niet uniek in Scratch: veel systemen werken zo.

In hoofdstuk 7 onderzoeken we of we het uitvoeringsmodel van Scratch zo kunnen wijzigen dat stappen door de code mogelijk wordt zoals hierboven beschreven (één blokje in elke stapel per stap), zonder negatieve effecten op de snelheid en het gedrag van bestaande Scratch-projecten. Aangezien Scratch zoveel gebruikt wordt, kunnen we geen wijzigingen voorstellen die ervoor zorgen dat een groot deel van de bestaande projecten stopt met werken of zich anders gaat gedragen. Om hier met kennis van zaken over te kunnen oordelen, hebben we eerst onderzocht hoe een typisch Scratch-project er in het wild uitziet. Hieruit blijkt dat de meeste Scratch-projecten klein en eenvoudig zijn.

Tot slot sluit hoofdstuk 8 dit proefschrift af door al ons werk, dat we in de verschillende hoofdstukken uit de doeken deden, samen te vatten en te overpeinzen wat de toekomst kan brengen.

Summary

Learning to program is hard, and many students find programming courses hard. As the idiom tells us, practice makes perfect. This is no different in programming education: it is generally accepted that the best way to learn programming is through experience. However, to actually learn something from these experiences, qualitative and timely feedback is crucial.

Yet providing this feedback on many exercises for many students is labour-intensive and time-consuming. This is why there is a long history (since at least the early 1960s) of using automation to provide feedback. The process of providing this feedback is called automated assessment. In most cases, automated assessment for programming education involves software testing. The code written by the students for a certain exercise (we call this a submission) is tested for at least correctness. Often, the feedback is more detailed than just a global correct or wrong.

Long is relative of course, but appropriate considering programming education itself debuted in the early 1960s.

As many have done before us, our department also created an online platform for automated assessment: Dodona. One of its key features is the separation between the platform itself (responsible for user management, course management, the user interface, etc.) and the judge (the testing framework responsible for evaluating submissions). Consequently, Dodona can support almost any programming language. It currently supports C, Haskell, Java, Kotlin, Prolog, R, Scheme, Bash, C#, JavaScript, Python, HTML, SQL, Markdown, and Turtle.

While working on and with Dodona, we observed some shortcomings in existing educational tools for with programming education. A more detailed look at the educational context and the Dodona platform is given in chapter 1. In summary, this dissertation attempts to overcome five of these observed shortcomings.

We observed that a lot of exercises in Dodona are suitable for use in multiple programming languages, at least in theory. To actually use them with another programming language, one must first copy the exercise, then manually convert the test suite to whatever format is used by the judge in the target language, and finally change the configuration files and task descriptions. This is a lot of manual work. Chapter 2 provides a

solution: **TESTed**, an educational software testing framework. Its defining feature is the support for creating programming-language-agnostic exercises. This means that one exercise (with a single test suite) can be solved in multiple programming languages, with support for automated assessment. An exercise is thus usable in different programming languages without any additional work.

With the TESTed prototype in hand, we then took a step back to look at what is required to go from a prototype to a viable option for creating programming exercises. We wanted TESTed to be the default option for creating programming exercises for Dodona. As such, it needs to be suitable for educators in both higher and secondary education. This resulted in the creation of **TESTed-DSL**, presented in chapter 3: a domain-specific language for authoring programming exercises with support for automated assessment across programming languages. A domain-specific language is a format or language specifically designed for one use case, which is authoring programming exercises here. It turns out that by paying special attention to the ergonomics of TESTed-DSL, it is also suitable for exercises that are not intended to be used in multiple programming languages. For example, we now also recommend TESTed for educators looking to author programming exercises that target JavaScript.

Teaching programming to young children is often done differently than teaching older students, by using visual programming languages. A visual programming language lets users create programs by manipulating program elements graphically, rather than textually. The most popular educational one of these is Scratch. In Scratch, programming consists of dragging blocks around and clicking them together (not unlike puzzle pieces or Lego bricks). Since Scratch works with blocks, it is also called a block-based language. A more detailed introduction to Scratch can be found in chapter 4.

Since Dodona supports multiple programming languages, we initially created a judge (the testing framework) for Scratch within Dodona. However, Scratch is not just a programming language, it is also a programming environment. It thus became clear that the needs for a platform that supports Scratch were too different from what we can do in Dodona. For this reason, we partnered with CodeCosmos, an industrial/commercial partner. As they are an educational publisher whose products include Scratch exercises, they already have a platform for working with Scratch. Additionally, they have more experience in creating exercises for Scratch.

Chapter 5 presents **Itch**, our testing framework for Scratch. It supports both static tests (meaning a test only looks at the blocks of the program without executing it) and dynamic tests (where the program is executed with some inputs and the results are observed). The combination of both

means Itch can test a wide variety of Scratch programs. Scratch is rather game-like and exploratory, which encourages children to experiment and use their fantasy. This does introduce challenges when attempting to test Scratch programs. For example, if the instructions are "Draw a house", how can we verify if the program is correct? Consequently, we do have to place some limits on what types of exercises Itch can test. The considerations that go into these decisions are also explained in the chapter.

When a testing framework like Itch gives feedback to students, everything is sometimes correct, but more often than not, some test cases fail. At that point, the debugging process begins: the students have to figure out what the cause of the failed test is. This is notoriously difficult since the location of the cause in the program is often not obvious. However, there are tools to help with this, the main tools being debuggers. For textual programming languages, there are a lot of debuggers and a lot of research into debuggers. As an example, Dodona supports a web-based debugger for Python.

However, for Scratch and block-based languages in general, this is not the case. Therefore, we introduce a new debugger for Scratch in chapter 6: **Blink**. Blink supports stepping through the code (i.e. going one step at a time when running the program), pausing and resuming the execution of a program, breakpoints (special blocks that automatically pause the execution when they are executed), and time travelling. A time-travelling debugger allows going backwards in the execution by recording program execution. Every step of the program is saved, so we can go back step-by-step. Since Scratch is used mainly by a young audience, we took special care to make the debugger intuitive. Initial tests in the classroom show that students find the debugger intuitive, especially its time-travelling feature.

In the previous paragraph, we said the debugger allows going one step at a time in the program. However, we did not specify what a step means in the context of Scratch. In Scratch, a project (the program) consists of different sprites (which are drawn on the screen). Each sprite has its own code, a set of scripts (a script is a set of connected blocks). Every script from every sprite can be run concurrently in Scratch. Consequently, we believe a traditional step in a debugger (advancing one block in a single script from one sprite) is not ideal. We want the step feature to advance a single block in every running script across all sprites.

However, due to the way Scratch works internally (the execution model), advancing a single block in every script is not possible. Scratch uses an almost-cooperative threading model, which means it executes multiple blocks in the same script, then jumps to the next script, and so on. By

Concurrency by fast switching is not unique to Scratch: a lot of concurrency works like this.

quickly switching between scripts, it looks like scripts execute in parallel. While this system is designed to prevent some concurrency-related issues, it causes other issues resulting in some unintuitive behaviour.

In chapter 7, we investigate if we can modify the Scratch execution model in such a way that we can implement the stepping functionality as described above (one block in every script per step), without negatively affecting performance and behaviour in existing Scratch projects when executing normally. Since Scratch is so widely used, we cannot introduce changes that would cause a large part of existing projects to stop working or behave differently. To properly evaluate this, we also look into what a typical Scratch project in the wild looks like. It turns out that most Scratch projects are simple and small.

Finally, chapter 8 concludes this dissertation by summarizing the work we presented in the various chapters and by reflecting on potential future endeavours.

Dankwoord

Het gebeurt wel eens dat een dankwoord van een doctoraatsproefschrift begint met de vermelding dat het om een werk van lange adem gaat. Dat is echter niet hoe ik het ervaren heb: aan de start van mijn doctoraat leken de vier jaar een eeuwigheid, maar nu heb ik het gevoel dat alles heel snel gegaan is. Ongetwijfeld heeft de steun van collegae, vrienden en familie hier een groot aandeel in: iedereen weet dat de tijd sneller gaat als het leuk is. Dat brengt ons bij het doel dezes paragraaf: mijn dank uiten aan een hele hoop mensen.

Ik negeer dat de tijd ook sneller gaat naarmate we ouder worden.

Allereerst wil ik mijn promotoren bedanken: prof. Peter Dawyndt, prof. Bart Mesuere en prof. Christophe Scholliers. Van alle mensen vermeld in dit dankwoord zijn zij waarschijnlijk degenen die de inhoudelijkste bijdragen geleverd hebben aan dit proefschrift. Dit gaat dan van de in regel wekelijkse vergaderingen waar we soms kort, soms lang bepaalde onderwerpen bespraken, tot de vele suggesties en opmerkingen over verschillende artikels (en dit proefschrift). Bedankt voor het mogelijk maken van dit doctoraat, en de voortreffelijke begeleiding en steun die ik hierbij gekregen heb.

Ik wil ook graag mijn juryleden (de "examencommissie") bedanken: prof. Bart Dhoedt, prof. Veerle Fack, prof. Gordon Fraser, and prof. Frank Neven. Een speciale bedanking ook aan prof. Chris Cornelis om de jury te willen voorzitten.

Special thanks to Gordon for coming all the way to Ghent for my internal defence.

Delen van dit proefschrift zijn tot stand gekomen in samenwerking met FTRPRF/CodeCosmos. Hoewel er soms iets misliep, zoals de judge die plots stopt met werken, ben ik trots op wat we samen verwezenlijkt hebben. Ik wil dan ook de mensen met wie ik heb samengewerkt bedanken: Katelijne Duerinck, Kristien Duerinck, Peter Keyngnaert, Cedric Vanhaverbeke, Elisa David, Morgane Kruglanski, Jannick Vandaele en Glenn Thielman.

Hoewel de meeste collega's geen rechtstreekse bijdragen geleverd hebben aan dit proefschrift, zorgden ze door een toffe werksfeer (en ook buiten het werk, zoals met het TWIkend) ervoor dat ik zin had om te blijven werken aan de Universiteit Gent. De lange lijst is Alexis Langlois-Rémillard, Asmus Bilbo, Benjamin Rombaut, Charlotte Van Petegem, Felix Van der

Als de soms wat te lange koffiepauzes in overweging genomen worden is de bijdrage misschien soms negatief.

jeugt, Heidi Van den Camp, Jonathan Peck, Jorg Van Renterghem, Louise Deconinck, Mustapha Regragui, Oliver Urs Lenz, Pieter Verschaffelt, Rien Maertens, Robbert Gurdeep Singh, Tibo Vande Moortele, Tom Lauwaerts, Toon Baeyens, Simon Reyntjens en Steven Van Overberghe. Veel van deze mensen waren al of zijn ondertussen ook vrienden geworden. Ik wil ook Annick Van Daele bedanken, voor het leuk maken van de werkcolleges Programmeren, samen met Charlotte en Toon.

Hoe verder in dit dankwoord, hoe verder van het bijdragen aan het proefschrift *in se* dat ik ga, waardoor ik nu aangekomen ben bij de vrienden die ik tijdens mijn opleiding heb leren kennen: Arne Gevaert, Jarre Knockaert, Jorg Van Renterghem, Louise Deconinck, Nils Mak, Pieter Verschaffelt, Rien Maertens, Sam Persoon, Sander Vanhove, ondertussen aangevuld met Chloë, Charlotte (Parmentier), Ciel en Liesbeth. Bedankt voor de vele leuke activiteiten we samen doen.

Pieter, ik denk dat ik niet anders kan dan je nog eens apart te vermelden. Sinds praktisch het begin van de opleiding, nu al tien jaar geleden, hebben we heel veel samen gedaan. Dit waren soms kleine dingen, zoals groepswerken, keuzevakken, examentoezichten of gaan eten bij de Griek. Soms waren het grotere dingen, zoals als we eens op reis gaan. Ik denk dat onze reis naar Malta en onze tocht door Bulgarije, Griekenland en Noord-Macedonië (samen met Jarre) me nog lang gaan bij blijven. Ik ben je dankbaar voor al deze momenten, en ondanks dat we nu voor het eerst in tien jaar niet meer samen studeren of werken, weet ik dat er nog veel leuke momenten gaan volgen.

Hydra is mijn langstlopende project aan de universiteit.

Nog wil ik de vele leden van ZeusWPI bedanken, waar ik tijdens de opleiding veel plezier aan gehad heb, en tijdens de eerste jaren van mijn doctoraat bij het thuiswerken vanwege de coronapandemie veel uren heb doorgebracht op Mattermost. De lijst van leden die ik zou kunnen bedanken is zo lang, dat ik er zelfs niet aan durf te beginnen. Daarnaast wil ik graag Wout Schellaert bedanken om mij te overtuigen een studentenbaan aan te nemen bij de Dienst StudentenActiviteiten. Ook alle medewerkers die daar destijds werkten wil ik bedanken, met een speciale vermelding van Nicolas Vander Eecken.

Tot slot, *last but not least*, wil ik mijn familie bedanken: mama, papa, Eliah en Maya, die mij allemaal wat meer of minder geholpen hebben tijdens mijn doctoraat, hetzij door te helpen verhuizen, hetzij door een luisterend oor te zijn als het even minder ging, hetzij door met mij op reis te gaan. Mijn ouders in het bijzonder ook om mij alle kansen te geven en mij te steunen in de keuzes die ik maak, zoals de keuze om informatica te studeren. Bedankt!

Een proefschrift als het mijne is vaak zeer eenzijdig: ik, als auteur, vertel allerlei dingen en van de lezer verwacht ik hoogstens wat denkwerk. Hoog tijd om daar verandering in te brengen! In onderstaande woordzoeker komt elke voornaam van elke vermelde persoon uit dit dankwoord voor. Als oplossing geeft hij nog mijn laatste boodschap aan de lezer voor de echte inhoud begint.

V	E	E	B	A	R	T	N	S	S	L	M	A	M	A	C	V
P	P	T	U	O	W	O	T	I	I	A	S	M	U	S	E	L
E	G	E	Z	I	O	E	M	E	R	X	M	I	M	E	D	E
E	O	R	T	T	V	O	T	R	A	B	E	N	R	A	R	F
L	B	R	O	E	N	T	X	I	L	E	F	L	H	H	I	R
I	I	A	N	J	R	L	E	T	Z	E	E	N	A	E	C	A
S	T	J	S	A	N	D	E	R	R	N	N	N	I	S	M	N
A	N	I	M	A	J	N	E	B	E	E	I	N	L	I	U	K
E	V	N	Y	A	A	N	M	H	L	C	B	L	E	U	S	E
N	IJ	A	E	G	R	N	P	G	O	I	H	B	S	O	T	T
IJ	M	P	R	I	E	O	R	L	J	G	E	L	O	L	A	T
L	J	O	N	A	T	H	A	N	O	A	O	S	O	R	P	O
E	M	E	F	S	E	S	S	R	C	H	N	R	B	E	H	L
T	O	R	I	I	I	I	I	P	A	P	A	N	D	E	A	R
A	T	R	D	F	P	E	T	R	E	T	E	P	I	O	T	A
K	H	I	A	N	N	I	C	K	K	L	E	I	C	C	N	H
C	H	A	R	L	O	T	T	E	R	E	V	I	L	O	K	C

Een kleine aanwijzing: er worden 51 mensen vermeld in dit dankwoord en het antwoord bestaat uit 38 letters.

Table of contents

List of publications

Included in this dissertation

List of publications directly related to this dissertation. As such, the contents of these publications are included in this dissertation.

Niko Strijbol, Charlotte Van Petegem, Rien Maertens, Boris Sels, Christophe Scholliers, Peter Dawyndt and Bart Mesuere (May 2023). "TESTed: An Educational Testing Framework with Language-Agnostic Test Suites for Programming Exercises". In: *SoftwareX* 22, p. 101404. ISSN: 2352-7110. DOI: 10.1016/j.softx.2023.101404.

Niko Strijbol, Robbe De Proft, Klaas Goethals, Bart Mesuere, Peter Dawyndt and Christophe Scholliers (Feb. 2024). "Blink: An Educational Software Debugger for Scratch". In: *SoftwareX* 25, p. 101617. ISSN: 23527110. DOI: 10.1016/j.softx.2023.101617.

Niko Strijbol, Boris Sels, Charlotte Van Petegem, Rien Maertens, Christophe Scholliers, Bart Mesuere and Peter Dawyndt (2024). "TESTed-DSL: A Domain-Specific Language to Create Programming Exercises with Language-Agnostic Automated Assessment". In: *Software Testing, Verification & Reliability*. Manuscript submitted for publication.

Remainder

List of publications by Team Dodona for which I am a co-author.

Rien Maertens, Charlotte Van Petegem, **Niko Strijbol**, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt and Bart Mesuere (2022). "Dolos: Language-agnostic Plagiarism Detection in Source Code". In: *Journal of Computer Assisted Learning* 38.4, pp. 1046–1061. ISSN: 1365-2729. DOI: 10.1111/jcal.12662.

Charlotte Van Petegem, Louise Deconinck, Dieter Mourisse, Rien Maertens, **Niko Strijbol**, Bart Dhoedt, Bram De Wever, Peter Dawyndt and Bart Mesuere (Mar. 2023). "Pass/Fail Prediction in Programming Courses". In: *Journal of Educational Computing Research* 61.1, pp. 68–95. ISSN: 0735-6331, 1541-4140. DOI: 10.1177/07356331221085595.

Charlotte Van Petegem, Rien Maertens, **Niko Strijbol**, Jorg Van Renterghem, Felix Van Der Jeugt, Bram De Wever, Peter Dawyndt and Bart Mesuere (Dec. 2023). "Dodona: Learn to Code with a Virtual Co-Teacher That Supports Active Learning". In: *SoftwareX* 24, p. 101578. ISSN: 23527110. DOI: 10.1016/j.softx.2023.101578.

Charlotte Van Petegem, Kasper Demeyere, Rien Maertens, **Niko Strijbol**, Bram De Wever, Bart Mesuere and Peter Dawyndt (Apr. 2024). *Mining Patterns in Syntax Trees to Automate Code Reviews of Student Solutions for Programming Exercises*. Manuscript submitted for publication. arXiv: 2405.01579. Pre-published.

Rien Maertens, Maarten Van Neyghem, Maxiem Geldhof, Charlotte Van Petegem, **Niko Strijbol**, Peter Dawyndt and Bart Mesuere (May 2024). "Discovering and Exploring Cases of Educational Source Code Plagiarism with Dolos". In: *SoftwareX* 26, p. 101755. ISSN: 23527110. DOI: 10.1016/j.softx.2024.101755.

Chapter 1.

Introduction

> *Computer science education research has lately been gathering momentum. It is now a mainstream area of doctoral research. Professional conferences catering to it are increasing in number and ranking. ... It signals the maturing of computer science education.*
>
> — ACM, IEEE, AAAI, *Computer Science Curricula 2023*

Learning to program is hard, and students consequently regard programming courses as difficult (Robins et al. 2003; Simões and Queirós 2020). It is generally accepted that gaining a deep understanding of programming requires experience and feedback (Gomes and Mendes 2007; Hattie and Timperley 2007). This feedback is what makes teaching programming difficult: it is a time-consuming task to provide qualitative and timely feedback on submissions, especially if the number of students in a course is high (Gulwani et al. 2014; Pirttinen et al. 2018; Queirós and Leal 2011; Staubitz, Teusner et al. 2017; Tang et al. 2016; Zavala and Mendoza 2018).

1.1. Origins and use of automated assessment in computer science education

Education of concepts that we would now consider part of the computer science discipline goes back as early as the 1940s. Of particular interest to this dissertation is the education of programming, which began appearing in curricula during the early 1960s (Simon 2015). This is shortly after computer science became recognized as its own scientific discipline in the 1950s and 1960s (Atchison 1971; Denning 2013; Gorn 1963; Hopcroft 1987; Knuth 1974). A good overview on the history of computer science education is given in Tedre et al. (2018), to which we point the reader for a complete overview.

From the start of programming education, the need for automated assessment was noted. Generally considered to be the first publication on automated assessment, Hollingsworth (1960) details their testing framework and remarks that they could not accommodate the number of students in their programming classes without their "automatic grader". Since then, there has been a long history of using automated assessment (Ala-Mutka 2005; Combéfis 2022; Douce et al. 2005; Ihantola et al. 2010; Messer et al. 2024; Nayak et al. 2022; Paiva, Leal et al. 2022).

In its most basic form, automated assessment for programming exercises translates to using some (educational) software testing framework that will evaluate the submitted code (the submission). The testing framework determines if a particular submission is correct, based on a set of tests (the test suite) provided by the educators who created the programming exercise. The result of this evaluation is the feedback given to the students.

There is a significant overlap between the software testing frameworks used in programming education and the ones used in programming contests, where these frameworks are called (online) judges, a term introduced by Kurnia et al. (2001). The International Collegiate Programming Contest (ICPC) is considered to be the oldest and most widely known programming contest. It traces its roots back to the First Annual Texas Collegiate Programming Championship, held in 1970 at Texas A&M University. From 1977 until 2017, the contest was held under the auspices of the Association for Computing Machinery, and known as ACM-ICPC (*ICPC Fact Sheet* 2023).

In essence, a testing framework used in educational practice and a competition must do the same thing: evaluating a submission for correctness. It is therefore not a surprise that judges are heavily used in educational contexts (Liu et al. 2023; Wasik et al. 2018; Zinovieva et al. 2021). The differences are mostly found in the focus of the testing frameworks. In educational settings, evaluating the correctness of a submission itself is not the main goal. The goal is to provide students with formative feedback (Caiza and del Alamo 2013; Cavalcanti et al. 2021). Even more, correctness as the only feedback might not help students in the learning process (Hao et al. 2021). In a programming contest, the result of an evaluation is often limited to a binary correctness decision: the submission is accepted or rejected. A second difference is the focus on performance: in an educational setting, the focus is generally on the performance of generating the feedback, while in a competition the focus lies on the performance of the submission. For example, in some programming contests, a correct submission is expected, but the competition is having the fastest submission.

Another area of interest is the practice of software testing in the software engineering field. As with educational software testing, the essential purpose of software testing is correctness (Pan 1999). The desired or correct behaviours are specified as the functional requirements of the program and say how a program must behave (Bass et al. 2021). Other software quality factors (the non-functional requirements) that may be tested are its functionality (reliability, usability, integrity), engineering (efficiency, testability, documentation, structure) and adaptability (flexibility, reusability, maintainability) (Hetzel 1988).

What makes educational software testing different is that each programming exercise has a fixed specification, against which multiple submissions must be evaluated (Wilcox 2016). Submissions are usually small to moderate in size, with all source code contained in a single file. While educational software testing also evaluates submissions for correctness, correctness itself is again not the main goal. Instead, the goal is to provide formative feedback (or a grade) to the students (Caiza and del Alamo 2013).

1.2. The Dodona platform

In this section, we discuss our programming course management system. For a full overview of these systems, we point the reader to Messer et al. (2024). Some noteworthy ones are *Mooshak*, *UVa Online Judge*, *Sphere Online Judge*, *Kattis*, *Jutge.org*, and *ArTEMiS* (Enstrom et al. 2011; Kosowski et al. 2008; Krusche and Seitz 2018; Leal and Silva 2003; Petit et al. 2018; Revilla et al. 2008).

As mentioned in the previous section, providing qualitative and timely feedback to students in programming courses almost necessitates a form of automation. This is no different for courses taught by our department. Feedback was initially (since 2011) provided using the Sphere Online Judge (Kosowski et al. 2008), which was relatively unique in that it allowed teachers to create their own courses, exercises, and even testing frameworks. However, the platform targets the organization of programming contests and lacks support for features we wanted to introduce in our programming classes.

In 2016, the needs for these classes outgrew the Sphere Online Judge, which led to the creation of our own platform: Dodona (Van Petegem, Maertens et al. 2023). Dodona is now used by most higher education

As a student, I was part of one of the last graduation years to use SPOJ (that statement makes me feel old).

institutions in Flanders and by many schools providing secondary education. As of August 2024, Dodona has more than 70 000 registered users and has evaluated 18.5 million submissions.[1]

The need for supporting multiple programming languages has also been noted from almost the start of using automated assessment for programming exercises (Hext and Winings 1969). Dodona has been developed with support for multiple programming languages from the start. It has a strict separation between the platform (responsible for managing courses, students, and exercises) and the testing framework (determining if submissions are correct). This separation has made supporting a multitude of programming languages relatively easy. Within Dodona, a testing framework is called a judge (and evaluating a submission for correctness is called judging), as it is called in the Sphere Online Judge.

While the earliest Dodona judges targeted Python and JavaScript submissions, its interface for judges proved to be generic enough that it now supports a multitude of scenarios. For example, there are currently judges for C, Haskell, Java, Prolog, R, Scheme, Bash, C#, JavaScript, and Python. There are also a few less straightforward judges, for HTML, SQL, Markdown, and Turtle.

1.2.1. Architecture

The separation between the Dodona platform and its judges is also reflected in its architecture. The platform itself is a fairly standard web application written in Ruby on Rails. The web application follows the recommendations by Rails: it uses a model-view-controller architecture. Most pages, especially simpler ones, use server-side rendering. More recently, complex pages have been added that use web components.

Evaluating submissions has some performance and security considerations (Wasik et al. 2018), as a submission is untrusted code that is executed. Dodona solves this by running the evaluation completely inside a dedicated Docker container (Peveler et al. 2019).

The Docker containers run on a dedicated pool of worker servers. Using dedicated workers, a misbehaving submission has no effect on the stability of the platform itself. Using worker servers also allows for easy scaling if the number of submissions increases.

[1] `https://dodona.be/en/about/`

At a high level, an evaluation of a submission goes through the following process:

1. The submission is saved, and a job is started to perform the evaluation.
2. A predefined and judge-specific Docker image is used to create a Docker container. In this container, all dependencies for the judge are available.
3. A temporary directory is created by Dodona containing the submission, the test suite for the relevant exercise, and (optionally) additional exercise-specific resources.
4. This folder is mounted in the container.
5. The judge runs inside the Docker container, using a well-defined interface containing some metadata for the judge.
6. During the evaluation, the judge outputs the feedback in a well-defined format on standard output.
7. The output is captured and saved, after which the feedback is shown on Dodona.

1.2.2. Features for educators

Besides automated assessment, Dodona provides a host of features to support educators when organizing programming courses.

To start, Dodona supports a sophisticated course and user management system. Educators can request that their account be upgraded to a teacher account, which grants access to additional capabilities, like creating and managing courses, but also access to the sample solution for exercises. Courses on Dodona are created to reflect courses in real life, and are linked to an academic year. Students can then subscribe to their relevant courses. A course consists of a number of series, which in turn contain some exercises, allowing for the creation of learning paths in the course.

Within a course, Dodona provides detailed statistics and data visualizations of the progress students make in a course and for each series. Dodona has a large amount of data about students, which has also allowed for interesting research. For example, Van Petegem, Deconinck et al. (2023) attempt to predict if students are at risk of failing the course, which would allow early intervention by the teaching staff.

Besides the automated feedback, Dodona allows students to ask questions on their submissions (if enabled by the educator for a specific course). This allows the teaching staff to answer these questions directly

in Dodona, with all the relevant context (the submission itself and the feedback the students got). At the same time, the submission history of the students is also available.

The same underlying mechanism used for the questions can also be used by educators to add annotations to submissions. This is especially useful in Dodona's "evaluation mode", which is designed to facilitate grading of submissions. Besides adding annotations to submissions (we are researching if predicting these annotations is possible; Van Petegem, Demeyere et al. 2024), assigning a grade to a submission using a rubric is also possible. We use this mode ourselves to organize exams as follows. A special series is shared with students at the start of an exam. This series contains the exercises that students need to solve. After the deadline, Dodona can automatically select the latest submission before the deadline for each student. We then manually provide feedback and grades on these submissions. After we have processed all submissions, we release the feedback to the students, who can then view the feedback and grade they received.

1.2.3. The feedback table

One of the most important actionable components for both students and educators is the feedback table (figure 1.1). This feedback table is flexible enough to accommodate a wide variety of feedback. In the example, three types of feedback are illustrated: the difference between the expected and generated return value, a custom message, and an image.

As the feedback table shows custom data from an exercise or judge, which are not necessarily trusted, special attention has been paid to make the feedback table secure. For example, all content is rigorously sanitized to prevent cross-site scripting attacks (S. K. Gupta and B. B. Gupta 2017).

1.2.4. Exercises and content management

Dodona currently supports two types of "learning activities": programming exercises and reading activities. A reading activity requires no submission, but students can mark them as read to confirm they did the reading. Programming exercises allow students to submit code and receive feedback on their submissions. However, the bar for what constitutes code can be low. For example, Dodona has a Markdown judge,

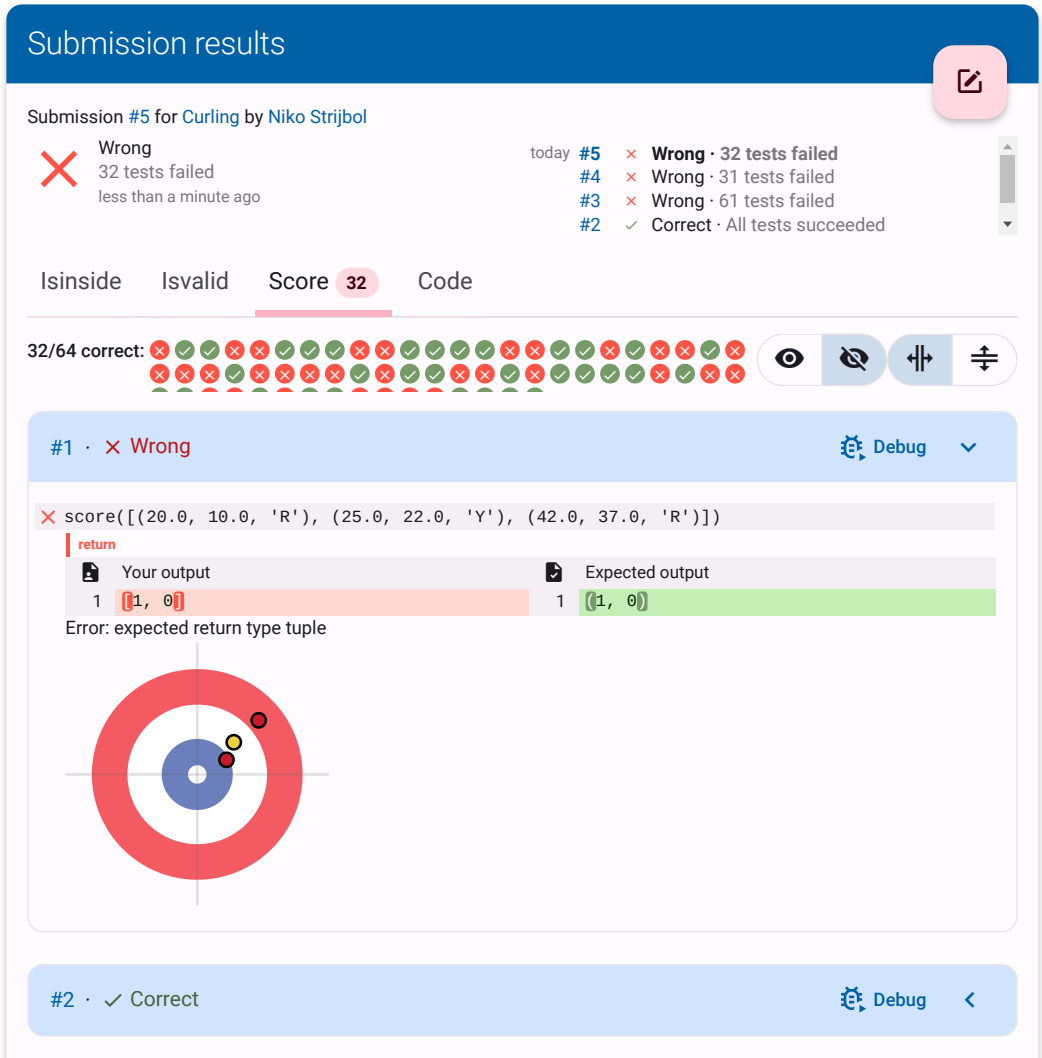


Figure 1.1. An example of the feedback table for an incorrect Python submission for the exercise *Curling* on Dodona. The feedback is split into three tabs (the fourth tab, "Code", contains the submission itself). As can be seen below the "Score" tab, only 32 of the 64 test cases are correct. The figure shows the first two test cases from the "Score" tab, the first of which is incorrect: the function should have returned a `tuple`, but returned a `list` instead. Besides showing the difference between the expected and actual value, Dodona also allows showing more information. In this case, a visual rendering of the curling position is given to aid students in debugging their code.

which generates a rendered (HTML) version of the Markdown code submitted by students, without automatically evaluating anything more than the validity of the Markdown. Dodona can thus be used to collect plain text submissions from students. Learning activities fully support internalization: Dodona currently supports Dutch and English.

Exercises in Dodona are managed in Git repositories. A repository with a well-defined structure is usable as the source for exercises in Dodona.[2] This allows educators to be the owners of their exercises, in addition to all the other benefits of version control. Every time an exercise is updated in the repository, Dodona will synchronize the exercises on the platform with the exercises in the repository.

Besides a test suite, which is needed for automated feedback, programming exercises also need a problem statement that specifies what students should do. In Dodona, a problem statement is a Markdown or HTML file, in which educators have free rein to describe their exercise. The programming exercise (including a problem statement and test suite), together with a judge (the Docker image and the testing framework) form a *task package* as defined by Verhoeff (2008).

1.3. Structure of this dissertation

The main problem this dissertation addresses is the observation that there are shortcomings in existing educational tools for programming education. Most of these shortcomings were first observed when using Dodona. They can be split into five main research questions:

RQ1 Can we design an educational software testing framework that supports automated assessment across programming languages based on a single test suite?

RQ2 What is the most ergonomic way to author programming exercises with support for automated assessment across programming languages?

RQ3 Can we design an educational software testing framework for the block-based programming language Scratch?

RQ4 Can we design an (educational) debugger for the block-based programming language Scratch?

RQ5 What common execution model for running and debugging Scratch code best optimizes both scenarios?

[2] `https://docs.dodona.be/en/references/repository-directory-structure/`

The first two questions relate to textual programming languages, while the last three questions are in the context of Scratch, a visual block-based language. The reason for this split is that it became clear that the needs for textual and block-based programming languages are different enough to warrant a separate approach.

This split is reflected in the structure of this dissertation as well. Part I is about textual programming languages and consists of two chapters (chapters 2 and 3), which answer the first and second questions respectively. Section 1.4 gives an introduction to this first part. Part II is about the block-based programming language Scratch. The smaller chapter 4 gives a short introduction to Scratch. Chapter 5 addresses question three, while chapter 6 address the fourth question. Chapter 7 then answers the fifth question. Section 1.5 gives an introduction to this second part.

Finally, chapter 8 summarizes the answers to the research questions and highlights some future opportunities. Each chapter also contains its own conclusion where the details for that specific chapter are discussed.

The remainder of this chapter gives a high-level overview of each part: the context, motivation, and where applicable, our previous publications. Similar to the conclusions, each chapter also has its own introduction, where the specifics for that chapter are introduced.

1.4. Textual programming languages

The first research question (RQ1) was inspired by the observation that using exercises and judges in Dodona required some manual work: "Can we design a testing framework that supports automated assessment across programming languages with a single test suite?". A lot of programming exercises on Dodona seem usable in multiple programming languages, at least in concept.

Actually using an exercise in another programming language first requires a copy of the exercise to be made. Then the existing test suite must be converted manually to the test suite format used by the judge for that particular programming language. The other parts of the exercise (the configuration and task description) also need to be converted manually. Additionally, since the exercises are copied, the exercises have to be kept in sync manually, or they risk diverging.

Implementing judges for different programming languages is also repetitive. For each judge, a new test suite format has to be created, and all other tasks that a judge does (e.g. test scheduling, output handling)

have to be re-implemented for each new judge. Supporting a new feature also requires doing it for every judge, which often does not happen in practice.

1.4.1. Programming-language-agnostic exercises

The answer to the first research question (RQ1) is TESTed, an educational software testing framework we created. Its defining feature is the ability to create programming-language-agnostic exercises. This means that the same exercise (with one test suite) can be solved in multiple programming languages, with support for automated assessment.

A doctoral dissertation has no page limit, after all.

TESTed is discussed in chapter 2, which is based on Strijbol, Van Petegem et al. (2023). Compared to the original publication, the chapter has been expanded with more information about the inner workings of TESTed, in addition to going more in-depth on technical aspects of the framework.

1.4.2. Ergonomic authoring of exercises

With a working prototype at hand, the second research question (RQ2) came rather naturally: "What is the most ergonomic way to author exercises with support for automated assessment across programming languages?". We took a step back to look at what is required to go from a prototype to a viable option for creating programming exercises. This analysis aimed to ensure the exercises could be used in educational practice, including high-stakes tests such as exams. We want TESTed to be suitable for both educators in higher education and secondary education. Our ambition was to make TESTed the default option for creating programming exercises in Dodona.

The result of this research is presented in a second publication: Strijbol, Sels et al. (2024). It is included almost verbatim as chapter 3. By looking at (educational) software testing more broadly, we find two missing parts of the process of creating programming exercises. The first missing and most important part is an ergonomic and approachable way of creating test suites for exercises. Our solution is TESTed-DSL, a domain-specific language for authoring programming-language-agnostic exercises with support for automated assessment. The second missing part is support for language-agnostic task descriptions. We show that TESTed-DSL can also be used for task descriptions.

A new insight while working on TESTed-DSL was that TESTed is not only useful for creating programming-language-independent exercises, but

is also suitable for exercises that are not intended to be used in multiple programming languages. Therefore, we have taken special care to design TESTed-DSL to be useful for a wide audience of educators, including higher and secondary education. For this reason, we also invested in our documentation, which contains reference documentation and a set of tutorials for commonly used exercise types.

1.4.3. Organization of the first part

Chapter 2 discusses TESTed and chapter 3 discusses TESTed-DSL, both based on previous publications. One consequence of this approach is that there is some overlap: for example, the introductions in both chapters broadly cover the same topic. However, the focus of both is different enough that we feel there is no problem including both. Another example of a consequence is the terminology used for the levels in the test suites (sections 2.6.1 and 3.2.1), which differs. The first chapter uses the terminology as used by Dodona, while the second chapter changes the terminology in the domain-specific language to align more closely with the terminology used in the literature. This illustrates the progressive insight that comes when working on a tool, and the interplay between the design stages and its application in practice.

Research on TESTed was started in 2019 as a master's thesis (Strijbol, Dawyndt et al. 2020), as was TESTed-DSL in 2021 (Sels et al. 2021).

1.4.4. Repositories and user documentation

As a software project, the source code for TESTed is important, if not more important than this dissertation. Both TESTed and TESTed-DSL share the same repository, which is published under the MIT licence.[3] Two sets of documentation are available, aimed at a different target audience:

- The guides for educators wanting to create programming exercises.[4] Most of these guides are currently only available in Dutch.
- The reference documentation, for a more in-depth look into more technical subjects.[5] For example, this includes the documentation on how to extend TESTed to add support for additional programming languages.

[3] https://github.com/dodona-edu/universal-judge
[4] https://docs.dodona.be/en/guides/exercises/
[5] https://docs.dodona.be/en/references/tested/

1.5. Block-based programming languages

For young students, learning to program is often done with specialized programming languages. The most-used language in this context is Scratch, a visual block-based programming language (Resnick, Maloney et al. 2009).

Since Dodona is independent of any testing framework, we thus asked the third research question (RQ3): "Can we design an educational software testing framework for Scratch?". Our intention was to add support for Scratch to Dodona.

1.5.1. Testing Scratch code

The first prototype of Itch, our testing framework for Scratch, did just that: it added support for evaluating Scratch submissions in Dodona. However, Scratch is not only a programming language, but a complete programming environment (Maloney et al. 2010). We realized that properly supporting Scratch required accommodations that were too different from what Dodona could (or would) provide. Additionally, while our group had experience with programming exercises for text-based languages, we had much less experience with Scratch in an educational context.

To address these issues, we sought an industrial and commercial partner. We found this partner in CodeCosmos, the international brand of FTRPRF, which is an educational publisher that creates, among other things, teaching packs. Schools can use these to fulfil their obligations as part of the move to introduce more computational thinking in secondary education.

The work was divided as follows: Ghent University was responsible for the technical aspects of the automated feedback for Scratch exercises, while CodeCosmos provided the lessons, the educational support, and last but not least, actual students to use the teaching packs. Readers interested in knowing more might like to read an article about this collaboration in *Dare To Think*, Ghent University's online magazine.[6]

The testing framework Itch, the answer to the third research question, is described in chapter 5. Itch allows both static and dynamic tests on

[6] `https://www.durfdenken.be/en/research-and-society/coach-codi-boosting-tool-helps-children-become-independent-coders`

Scratch exercises, which provides a lot of flexibility to educators. Dynamic testing, in particular, allows for exercises that support (within limits) the exploratory nature of Scratch.

Itch test suites for Scratch exercises are written in JavaScript. While working with CodeCosmos on creating Scratch exercises, it became clear that the JavaScript test suites were a big hurdle for educators without a computer science background. As a lot of software testing frameworks are written in the same programming language as the code they test, we also explored this approach. The result of this is Poke, a prototype of a testing framework for Scratch with test suites written in Scratch, discussed in section 5.7. While technically feasible, there are still open questions regarding the practical use of this approach in an education setting.

1.5.2. Debugging Scratch code

The goal of educational software testing and automated assessment is to provide students with feedback on their code. However, the feedback in itself is not enough: we want students to be able to use that feedback and be able to correct their code if something went wrong. The next step, after a failed test, is to debug the submission, which is a two-step process (Myers et al. 2012). Step one is determining the exact nature and location of the error, and step two is fixing said error.

It is well known that determining the cause of an observed failure is challenging (Ammann and Offutt 2016). The debugging process is difficult, especially for novice programmers (McCauley et al. 2008). This has led to the creation of various tools to help with this, chief among them debuggers (Rosenberg 1996).

This is also the case with text-based languages. For example, this is why figure 1.1 shows a "Debug" button in the feedback table of Dodona. However, for Scratch, this area is much less developed. The fourth research question (RQ4) thus became: "Can we design a debugger for Scratch?".

Chapter 6 discusses our answer to this question, in the form of Blink, our time-travelling debugger for Scratch. A time-travelling debugger allows the programmer to go back in the execution, often by recording program execution (Balzer 1969; Barr and Marron 2014; Barr, Marron et al. 2016; Chen et al. 2001; Crescenzi et al. 2000; Czaplicki and Chong 2013; Ungar et al. 1997). We took special care to make the debugger intuitive for the young target audience of Scratch. Initial experiments in the classroom show that children do in fact find the debugger intuitive

and useful, particularly stepping through the code and the time travel feature.

This chapter is a minimally modified copy of the publication Strijbol, De Proft et al. (2024).

1.5.3. The Scratch execution model

While working on the debugger, we also began looking more deeply into the Scratch execution model. The Scratch execution model combines a fixed-step time loop (30 frames per second) with an almost-cooperative threading model. This means that threads are seldom interrupted, mostly relying on explicit yielding to other threads.

This threading model was chosen because it minimizes the occurrence of some race conditions, although concurrency-related issues still occur (Maloney et al. 2010). The cooperative nature of the threading model also has drawbacks: it can lead to unexpected behaviour when working with multiple sprites.

Additionally, the execution model also limits how a debugger can work, without resorting to deviating from the execution model as used during normal execution.

We thus asked ourselves, "What execution model for running and debugging Scratch code best optimizes both?" (RQ5). We investigate and answer this question in chapter 7. The chapter begins with a detailed look at how the current execution model behaves, followed by our proposed changes. Since Scratch is so widely used, our changes should not negatively impact existing Scratch programs. We therefore analyse what a typical Scratch project looks like. Finally, we run some preliminary benchmarks with our proposed changes and report on the results.

While we initially set out to find *the* replacement for the execution model, we no longer believe that a single replacement is possible. Scratch programs, especially larger ones, rely on the current behaviour of the execution model, meaning any change to this behaviour would constitute a breaking change. The proposed changes are still useful, but in more specialized contexts, instead of a general replacement. For example, the proposed changes enhance the use of the debugger. Smaller programs are much less affected by our proposed changes, and most programs are small.

1.5.4. Organization of the second part

The second part starts with chapter 4, which introduces Scratch, the programming language and environment. Next, chapter 5 discusses Itch and Poke, our Scratch testing frameworks. Chapter 6 then discusses Blink, our debugger for Scratch. Finally, chapter 7 discusses the Scratch execution model, our proposed changes to it, an analysis of Scratch projects, and an evaluation of our proposed changes.

Work on these various tools for Scratch often began as a master thesis (Cattoire et al. 2024; De Proft et al. 2022; Goethals et al. 2023; Mak et al. 2019; Voeten et al. 2023).

1.5.5. Repositories

The source code of Itch is not publicly available at this time due to the agreement with our partner CodeCosmos. The source code for Blink is available under the same licence as Scratch, the BSD 3-Clause "New" or "Revised" Licence.[7] We also host an online instance of the debugger.[8]

Our JavaScript implementation of an analysis tool for Scratch 3.0 (similar to Hairball; Boe et al. 2013) is available under the MIT Licence.[9]

[7] https://github.com/scratch-ed/blink
[8] https://scratch.ugent.be/blink/
[9] https://github.com/scratch-ed/scratch-analysis

Part I.

TESTed

Chapter 2.

An educational testing framework with language-agnostic test suites for programming exercises

Testing leads to failure, and failure leads to understanding.
— Attributed to Burt Rutan

In educational contexts, automated assessment tools are commonly used to provide formative feedback on programming exercises. However, designing exercises for automated assessment tools remains a laborious task or imposes limitations on the exercises. Most automated assessment tools use either output comparison, where the generated output is compared against an expected output, or unit testing, where the tool has access to the code of the submission under test. While output comparison has the advantage of being programming language independent, the testing capabilities are limited to the output. Conversely, unit testing can generate more granular feedback, but is tightly coupled with the programming language of the submission.

In this chapter, we introduce TESTed, which enables the best of both worlds: combining the granular feedback of unit testing with the programming language independence of output comparison. Educators can save time by designing exercises that can be used across programming languages. We begin by giving a brief introduction and considering some related work.

We focus next on the technical aspects of TESTed, beginning with its architectural design. Next, we consider the three parts of evaluating a submission: the test suite, the evaluation process itself, and the generated feedback.

For the test suites, we discuss their structure, and look at how TESTed handles datatypes (across different programming languages), statements, and expressions. Next, the evaluation process is discussed in detail.

Note that TESTed now supports a user-friendlier test suite format: TESTed-DSL (chapter 3).

This is the process through which TESTed executes the test suite and determines the feedback that will be given on the submission. We also briefly look at the integration with Dodona. This is relevant, as TESTed inherits some aspects from Dodona, such as the feedback format. As a conclusion to the technical part, we discuss in detail how support for multiple programming languages is achieved. We look at the internal API to give a good idea of what is required to support new programming languages.

Finally, we evaluate TESTed in educational practice. We report on three quasi-experiments where we asked students to solve a set of programming exercises and automatically reviewed their submissions with TESTed.

2.1. Introduction and background

Formative feedback on solutions for programming exercises is a crucial part of learning to code (Luxton-Reilly et al. 2018; Orrell 2006; Shute 2008). Feedback is formative (and most valuable) if learners receive rich and qualitative feedback throughout the learning process (Black and Wiliam 2009). Providing such feedback by hand is a challenge for educators since it is a time-consuming activity (Campos et al. 2012; Cheang et al. 2003; Edwards et al. 2008; Gulwani et al. 2014; Hao et al. 2021; Keuning et al. 2018; Pirttinen et al. 2018; Staubitz, Teusner et al. 2017; Tang et al. 2016; Zavala and Mendoza 2018). The challenge is further exacerbated by a large number of students who work on multiple exercises (Camp et al. 2017; Sax et al. 2017). Any laborious and time-consuming activity is a prime target for automation, and providing feedback is no different. The computer science education field therefore has a long history of applying software testing frameworks or general-purpose software tools (linters, formatters, etc.) on source code to automatically generate feedback (Edwards 2004; Hao et al. 2021; Keuning et al. 2018; Paiva, Leal et al. 2022).

Off-the-shelf tools such as linters are low effort for educators, but only provide generic feedback: there is no exercise-specific feedback. Software testing frameworks require an accompanying test suite for each individual programming exercise, which means they require a lot more effort to create. However, they can provide much more specific feedback. A test suite typically contains a set of tests that verify if the submission satisfies the requirements in the task description of the exercise. This is similar to the practice of unit tests or integration tests in the software engineering field. Besides general testing frameworks, computer science education also uses judge systems (Paiva, Leal et al. 2022; Wasik et al.

2018): specialized learning environments that provide rich feedback. Defining an adequate test suite for an exercise is part of the reason why creating programming exercises is so time-consuming (Gulwani et al. 2014; Pirttinen et al. 2018; Queirós and Leal 2011; Staubitz, Teusner et al. 2017; Tang et al. 2016; Zavala and Mendoza 2018).

An apparent solution to reduce the amount of time needed to create programming exercises is to create fewer exercises. To achieve this without reducing the number of exercises available to students, maximal reuse of existing exercises is necessary. Existing solutions to facilitate reusing exercises include standard formats in which exercises are written (Paiva, Queirós et al. 2020; Verhoeff 2008), open repositories with exercises (Staubitz, Teusner et al. 2017), or tools to convert between existing exercise formats (Queirós and Leal 2013). However, reusing exercises remains difficult, particularly when attempting to reuse exercises across programming languages. In most existing judge systems, exercises are tightly coupled to the programming language in which the test suite is written. Alternatively, test suites without such tight coupling impose stringent restrictions on exercise specifications, for example only reading from standard input and writing to standard output. Because of these restrictions, they often generate feedback of a lower quality than programming-language-specific test suites.

To reuse programming exercises written for another programming language, exercise designers first have to learn a new testing framework for that programming language. Then, they have to write a new test suite for the exercise according to the specifications of the new testing framework. Alternatively, due to the vast number of programming languages (Bissyande et al. 2013), it might be necessary to implement a new testing framework to support educational software testing, which is no small undertaking.

We focus on facilitating easy reuse of programming exercises across programming languages. Our contributions include defining requirements for a testing framework to support multiple programming languages, without imposing strict restrictions on programming exercises. We also introduce TESTed, a proof-of-concept of a system that satisfies these requirements. TESTed directly supports multiple programming languages for the same exercise, meaning the conversion of exercises to multiple testing frameworks is no longer needed. Exercise designers working on exercises for a single programming language can also benefit from TESTed, as it no longer forces them to learn new testing frameworks for each new target language. Finally, implementing support for new programming languages in TESTed is less work than implementing a

complete testing framework for each language, reducing the software development and maintenance cost dramatically.

2.2. Related work

Since the terminology related to automated feedback on programming exercises is not used consistently within the field, we begin by defining the terms used in this chapter. A programming exercise is the combination of a **task description** and a **test suite**. When students attempt to solve the exercise using the instructions from the task description, they create a **submission** for the exercise. The test suite is used by a judge system to **evaluate** the submission. This results in **feedback** that is shown to the student.

We consider it useful to split a judge system into a **judge platform** and a **testing framework**. A judge platform is a graphical user interface that allows students and educators to upload and store submissions, display feedback, organize exercises into courses, and so on. A testing framework is responsible for generating feedback, by executing and evaluating submissions based on a test suite. In existing literature, the distinction between these two is not always made, nor is it always relevant. For example, some review papers (Keuning et al. 2018; Wasik et al. 2018) or individual tools (Bez et al. 2014; Petit et al. 2018) consider the entire system as a whole. Other papers focus specifically on judge platforms (Gusukuma et al. 2020; Striewe 2016). However, we believe this distinction to be relevant, as evaluating submissions (the testing framework) has a separate set of challenges compared to judge platforms (for example, displaying feedback in a constructive way). This chapter focuses on testing frameworks.

In educational contexts, software testing frameworks typically use either **output comparison** (also known as **input/output testing**) or **unit testing** (Paiva, Leal et al. 2022) to evaluate submissions. Programming language support of testing frameworks generally depends on the approach they adopt. With unit testing, the test suite is often written in the same programming language as the submission, e.g. with tools based on xUnit (Meszaros 2007). Since the test suite is often written in the same programming language as the submission, it has full access to the submission. The test suite can use function calls, data structures (e.g. lists, maps), primitive data types (e.g. integers, strings), examine return values, exceptions, runtime inspections (e.g. reflection), etc. As a result, the test suite is tightly coupled to a specific programming language. Evaluating a submission for the same exercise in another programming

language would require a new test suite, often also involving another testing framework.

With input/output testing, the judge system imposes stringent restrictions on the programming exercise: only standard input, standard output, standard error, and files can be used for input and output. Prominent examples are the testing frameworks used in ICPC-style (*International Collegiate Programming Contest*, sometimes still referred to as ACM-ICPC) (*ICPC Fact Sheet* 2023) programming contests. While this does make the test suites independent of any programming language, all aspects of the submission that the test suite needs to check must be converted to a textual representation. Another disadvantage is the lack of granular testing: with these systems, tests are limited to the program as a whole. It is, for example, not possible to add distinct tests for different functions inside the same program without obligating students to artificially split the program.

Wanting to support multiple programming languages is not new: Hext and Winings (1969) describe an early automated assessment system that supports multiple programming languages. While it supports more than just input/output testing, it does not support a single test suite: every programming languages requires its own test code (the system was limited to two tests). The reporting of the results was, however, unified across programming languages.

2.3. Programming-language-agnostic testing frameworks

Intuitively, an ideal testing framework that supports multiple programming languages is a testing framework with the testing capabilities of unit testing and the programming language support of input/output testing. First, we define the two requirements a framework needs to satisfy to achieve this. We also look at the consequences of those requirements on the framework. We call a testing framework that satisfies these requirements a **programming-language-agnostic testing framework**.

The **first requirement** is that test suites must be programming language agnostic. A test suite must be usable to evaluate submissions in every programming language supported by the framework, without making changes to the test suite or adding languages-specific tests to it. As a consequence of this requirement, adding support for new programming languages to the testing framework must not demand changes to existing

test suites. All existing test suites must work with the new programming language without changes.

The **second requirement** is that the framework must have testing capabilities similar to those of unit testing frameworks. Some concrete examples of what should be possible are:

- Using standard input, standard output, and standard error.
- Calling functions implemented in submissions, and passing arguments to those functions.
- Evaluating values returned from function calls (with data type support).
- Creating objects (constructors) and manipulating them (methods).
- Capturing and evaluating other side effects such as exceptions, exit codes or files (or other persistent storage such as databases).
- Supporting exercises with deterministic and non-deterministic (e.g. random) behaviour.

Support for these two requirements should not have disproportionately large drawbacks. Therefore, we also consider the following soft requirements for a programming-language-agnostic testing framework:

- Authoring exercises using the framework should not be significantly more difficult than authoring exercises using a language-specific testing framework.
- Runtime and memory overhead should be minimal: the framework cannot be unacceptably slow or have a large memory footprint compared to a language-specific testing framework. Providing feedback to students in a timely manner is important.
- Submissions should follow their language-specific conventions as closely as possible. For example, supporting asynchronous and synchronous functions in JavaScript, or top-level functions should be implemented as a static function in Java, but not in Python or Kotlin that have proper support for top-level functions.
- Adding support for new programming languages to the framework should ideally be faster than implementing language-specific testing frameworks for those languages, due to the ability to reuse shared parts of the implementation.

The next sections introduce TESTed, our implementation of such a framework. We begin by showing an example of how TESTed is used, to give a good idea of what TESTed does. Afterwards, the various parts of TESTed are discussed in detail.

2.4. Using the framework

We start with a high-level tour of TESTed, focusing on how the framework can be used. To this end, we will use the following programming exercise as an example:

> Implement a function `remove_all_occurrences` that takes two arguments: a list *l* of integers and an integer *v*. The function must return a new list containing the same integers as list *l*, in the same order, but where all occurrences of integer *v* are removed.

A correct Python implementation for this exercise is:

```
1  def remove_all_occurrences(l, v):
2      return [x for x in l if x != v]
```

A test suite for this exercise could contain a number of function calls with different arguments and check their return values against an expected value. To keep things short, we limit ourselves here to a test suite with a single function call (listing 2.1).

When evaluating a submission using this test suite, TESTed will first translate the test suite into the programming language of the submission. Conceptually, each function call is converted into a test case:

```
1  # Python
2  remove_all_occurrences([1, 2, 3, 2], 2) == [1, 3]
```

```
1  // JavaScript
2  removeAllOccurrences([1, 2, 3, 2], 2) === [1, 3]
```

```
1  -- Haskell
2  (removeAllOccurrences [1, 2, 3, 2] 2) == [1, 3]
```

TESTed takes care of the differences between programming languages (syntax, default representations, and naming conventions). The actual test code generated by TESTed is more complex, as it must capture return values, account for exceptions, etc.

```
{
 "input": {
  "type": "function",
  "name": "remove_all_occurrences",
  "arguments": [
   {
    "type": "sequence",
    "data": [
     {"data": 1, "type": "integer"},
     {"data": 2, "type": "integer"},
     {"data": 3, "type": "integer"},
     {"data": 2, "type": "integer"}
    ]
   },
   {"data": 2, "type": "integer"}
  ]
 },
 "output": {
  "result": {
   "type": "sequence",
   "data": [
    {"data": 1, "type": "integer"},
    {"data": 3, "type": "integer"}
   ]
  }
 }
}
```

Listing 2.1. Snippet of a JSON test suite with a single test case that calls the function `remove_all_occurrences` with two arguments: *i*) a sequence containing the four integers 1, 2, 3 and 2, *ii*) the integer 2. The expected return value is a list containing the integers 1 and 3.

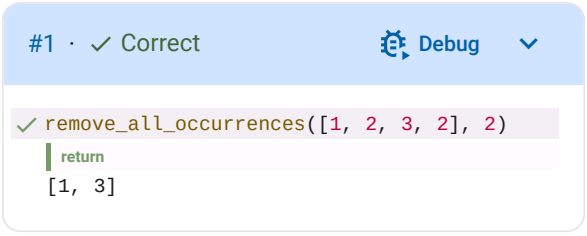


Figure 2.1. An example of how the feedback for the test suite from listing 2.1 can be rendered (in this case, it is rendered by Dodona).

When TESTed evaluates the correct Python submission from before, it produces the following Python-specific feedback:

```
{"command": "start-testcase"}
{"description": "remove_all_occurrences([1, 2, 3, 2], 2)"}
{"expected": "[1, 3]", "channel": "return"}
{"generated": "[1, 3]", "status": "correct"}
{"command": "close-testcase"}
```

A judge platform may then use this output to display a human-readable representation of the feedback (an example of this can be seen in figure 2.1). The feedback starts with a description of what has been tested (a function call in the programming language of the submission, in this case Python), followed by the expected return value and the actual return value (in a human-readable string representation). With the actual return value, a decision is also given: in this case, the return value is correct.

While the output for correct submissions is important, the output for wrong submissions is at least as important in the context of an educational testing framework. Assume a submission is wrong in that it only removes the first occurrence of the integer v from the list l (example in JavaScript):

```
function removeAllOccurrences(l, v) {
  l.splice(l.indexOf(v), 1);
  return l;
}
```

When TESTed evaluates this submission, its feedback reflects that while the code has been executed successfully, a logical error was found:

```
{"description": "removeAllOccurrences([1, 2, 3, 2], 2)"}
{"expected": "[1, 3]", "channel": "return"}
{"generated": "[1, 3, 2]", "status": "wrong"}
```

However, students sometimes do not even get to the point where their submission is executed successfully. For example, they might submit a solution containing a syntax error (in Python):

```
{"description": "remove_all_occurrences([1, 2, 2, 3], 2)"}
{"message": "Received compiler error:"}
{"message": "*** Error compiling..." }
{"status": "compilation error"}
```

Instead of an expected and actual return value, the feedback now contains the error message printed by the Python compiler. If compilation fails, the submission will not be executed and testing is stopped.

While Python is not considered a compiled language, it supports `python -m compileall`.

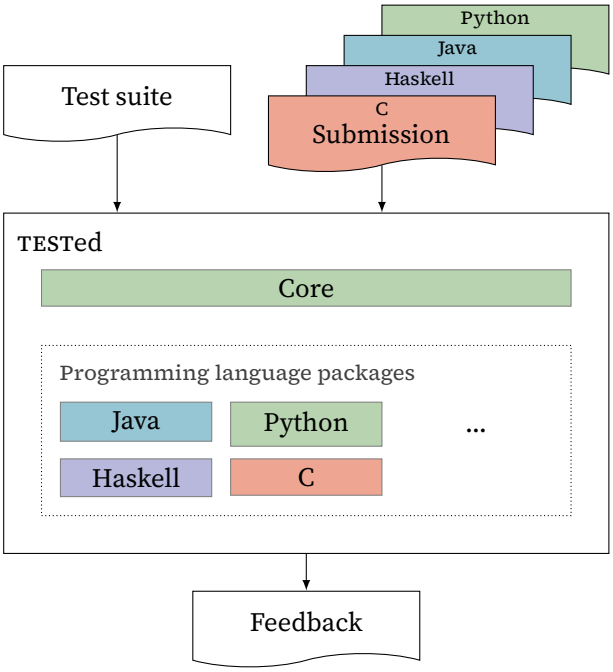


Figure 2.2. Architectural design of TESTed, with colours indicating different programming languages. The framework consists of a set of Python packages and modules. These can be categorized as the core package and a set of programming-language-specific modules. The input for TESTed consists of a test suite, together with a submission in one of the supported programming languages. The output is the generated feedback.

2.5. Architectural design of the framework

TESTed is built around the idea that an exercise author writes a single, unified test suite for the exercise, independent of any programming language. TESTed then converts that test suite into the programming language of the submission on the fly, and takes care of the various aspects of the evaluation process: compiling the submitted code, executing the submission together with the test code, checking test results, and generating feedback.

While some parts of the evaluation process are programming-language-specific by necessity, such as generating the test code, a lot of the steps are not. For example, creating an execution plan or interpreting the test results and generating the feedback are not specific to any one programming language. Therefore, the language-specific aspects are isolated in

language modules, as illustrated in figure 2.2.

TESTed is written in Python and organized into a set of Python modules and packages. An import package is the *core* package, which contains modules that are responsible for all language-agnostic tasks, such as scheduling tests. We provide an in-depth description of the full evaluating process in section 2.7.2. In most cases, the core is also responsible for checking the collected test results and generating the feedback. This is discussed in section 2.7.5.

A Python module is a `.py` file, while a Python package is a folder containing modules.

All aspects that are specific to one programming language are bundled in one package. These language-specific modules take care of all language-specific tasks, such as compiling submissions, executing submissions, and handling language-specific data types, expressions, and statements. Since the language-specific code is limited to these modules, this offers benefits for adding support for new programming languages to TESTed, as discussed in section 2.9.

TESTed takes as input the test suite for a programming exercise and a possible submission that needs to be evaluated. The format of the test suite is discussed in-depth in section 2.6. As a result of its evaluation, TESTed outputs a feedback report (section 2.8)

2.6. Test suites

Despite multiple proposals on generic formats for programming exercises (Edwards et al. 2008; Paiva, Queirós et al. 2020; Queirós and Leal 2011; Verhoeff 2008) or exercise classifications (Le et al. 2013; Simões and Queirós 2020), there seems to be no generally accepted standard for test suites of programming exercises. Additionally, none of the proposed formats meets the requirements that we identified for language-agnostic testing frameworks. Existing formats focus mostly on the task description, whereas we mainly focus on the specification of the test suite itself.

We therefore feel it is appropriate to introduce a new format for test suites, specifically developed for TESTed. It is designed to be machine and human-readable and easy to generate. Making it ergonomic to write by hand was only a secondary goal. While the format is not framework-specific, and could thus be adopted by other testing frameworks, this was not an explicit design goal.

Test suites for TESTed are written in JSON, which was chosen because there is broad support in programming languages. Additionally, it is

human-editable and readable, especially compared to binary formats. There is also widespread editor support for JSON, and technologies such as JSON Schema allow for easy validation and even better editor support.

Note that TESTed-DSL (chapter 3) is the intended way to author exercises, especially if done manually. The test suites discussed here are a direct JSON representation of the internal structure of test suites as used by TESTed, and thus much more verbose and not guaranteed to stay stable in future version of TESTed.

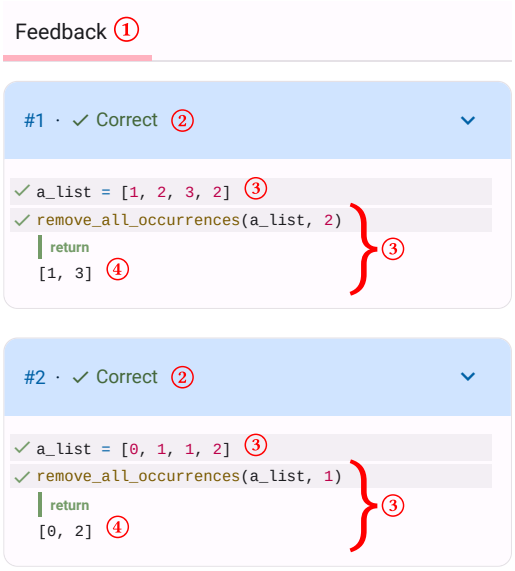
2.6.1. Structure of a test suite

Returning to the same programming exercise from before, we can expand the test suite of listing 2.1 for illustration purposes (resulting in listing 2.2, rendered in figure 2.3). Instead of directly calling the function `remove_all_occurrences` with two arguments, we first create a list and assign it to a variable in the first test case. In the second test case, we then call the function `remove_all_occurrences`, but use the variable name as the first argument.

While omitted for brevity in the examples, a test suite can organize test cases in two levels: tabs and contexts. The full hierarchy of the test suite consists of four levels, from top to bottom:

1. **Tabs** are the top-level grouping mechanism. It allows logically grouping contexts together. While the name of this group suggests how to display these groups, it is but a suggestion. The example only has one tab, but a test suite with test cases for multiple functions might, for example, have one tab per function.
2. **Contexts** are meant to group dependent test cases together. In our example, declaring a variable and using that variable must be done within the same context.
3. **Test cases** are the basic building blocks of the test suite. The "input" for a test case is what exactly is being checked. In the example, the first test case of each context has an assignment, while the second has a function call.
4. **Tests** are used for each type of output. TESTed currently supports return values, standard output, standard error, exceptions, generated files and exit codes. In the example there is one test for the return value. If a test is not defined (e.g. for standard error in the example), a sensible default is used. For example, the default test for standard output and error checks that there is no output, causing the evaluation to fail if there is unexpected output.

```
{
 "variable": "a_list",
 "type": "sequence",
 "expression": {
  "type": "sequence",
  "data": [
   {"data": 1, "type": "integer"},
   {"data": 2, "type": "integer"},
   {"data": 3, "type": "integer"},
   {"data": 2, "type": "integer"}
  ]
 }
}, {
 "input": {
  "type": "function",
  "name": "remove_all_occurrences",
  "arguments": [
   "a_list",
   {"data": 2, "type": "integer"}
  ]
 },
 "output": {
  "result": {
   "value": {
    "type": "sequence",
    "data": [
     {"data": 1, "type": "integer"},
     {"data": 3, "type": "integer"}
    ]
   }
  }
 }
}
```



Listing 2.2. *(left)* A snippet of a JSON test suite for TESTed with two statements: *i*) declaration of a variable (`a_list`) that is assigned a sequence containing four integers 1, 2, 3, and 2 and *ii*) call of the function `remove_all_occurrences` with two arguments: `aList` and the integer 2. The expected return value is a list containing the integers 1 and 3.

Figure 2.3. *(right)* A way to visually render the feedback (as done in Dodona) resulting from evaluating a submission (in Python) with the test suite from the left. There are four levels: ① tabs, ② contexts, ③ test cases, and ④ the tests. Here, each context consists of two test cases, the first of which has no explicit tests, while the second has one explicit test (the expected return value).

Table 2.1. Overview of the basic types of TESTed and their implementation in the programming languages currently supported by TESTed. Sometimes, another type is used instead, based on the value. For example, an integer that is too large for `int` in Java will become a `long`. A dash (-) is used to indicate that the programming language does not support this type.

TESTed	Python	JavaScript	Java	Kotlin	Haskell	C	Bash	C#
integer	`int`	`Number`	`int`	`Int`	`Int`	`int`	-	`Int32`
real	`float`	`Number`	`double`	`Double`	`Double`	`double`	-	`Double`
boolean	`bool`	`Boolean`	`boolean`	`Boolean`	`Bool`	`bool`	-	`Boolean`
text	`str`	`String`	`String`	`String`	`String`	`char*`	`text`	`string`
sequence	`list`	`Array`	`List`	`List`	`[]`	-	-	`List`
set	`set`	`Set`	`Set`	`Set`	-	-	-	`Set`
map	`dict`	`Object`	`Map`	`Map`	-	-	-	`Dictionary`
nothing	`None`	`null`	`null`	`null`	`Nothing`	`void`	-	`void`

This structure mirrors the output generated by TESTed (section 2.8). For example, the executed input for each test is also included in the output. A possible visualization of these levels is given in figure 2.3, which shows the output rendered in Dodona (Van Petegem, Maertens et al. 2023).

2.6.2. Data serialization

TESTed uses data serialization to convert between the language-agnostic format of the test suite, and the generated test code. Additionally, the same data serialization is used to convert return values from the submission into the language-agnostic format for checking (as described in section 2.7.5).

Each literal value is described by an object with two attributes: a value (e.g. the number `5.3`) and a data type (e.g. `real`). These attributes are combined into a JSON object with two fields. The value is encoded using the closest representation available in JSON. For example, a number is represented by a JSON number, and a string is represented by a JSON string. The data type of a literal value is more complex, since TESTed targets multiple programming languages that each support their own collection of data types. TESTed therefore defines a set of rules to denote data types and their support in programming languages. This allows TESTed to convert types between programming languages.

TESTed uses two categories of data types. The first category is a limited set of **basic types** that are abstract and map to concepts. Currently, the following basic types are supported:

`integer` An integer. The size of the integer is left undefined.
`real` A real number. The size and precision of the real is left undefined.
`boolean` A Boolean value.
`text` Textual data (e.g. strings). The intention is important here: for example, an ASCII character can be represented as both an integer or as text.
`sequence` An ordered sequence of values.
`set` An unordered collection of unique values.
`map` A collection of key-value pairs, where the keys are unique.
`nothing` A representation of "nothing", meaning no value.
`any` Any or unknown data type. This type is not usable in test suites, but is used to indicated return values of an unknown type.

When a test suite contains a literal value of a basic type, it will be serialized as an object of an actual data type in the target programming language. An overview of all basic types and their implementation is given in table 2.1. For example, a literal value with data type `map` will become a `Map` in Java and a `dict` in Python.

The second category consists of **advanced types**, which are more detailed or programming language specific. Each advanced type is associated with a basic type, acting as a fallback. For example, `int64` is an advanced type with the basic type `integer` as a fallback. If a programming language does not support a particular advanced type, the corresponding basic type will be used. For example, consider tuples. Many programming languages do not have direct support for tuples, but exercises using tuples can still be solved by using the corresponding basic type (sequence). More concrete, an exercise using the advanced type `tuple` can be solved in Java by using a `List`.

When adding a programming language, it is possible to disable this fallback for certain types. For example, JavaScript has no support for fixed precision numbers. This prevents TESTed from evaluating submissions in JavaScript if fixed precision numbers are used in the test suite. TESTed will generate an appropriate error in this case.

Currently supported advanced types are:

`int8/16/32/64` 8/16/32/64-bit integers
`uint8/16/32/64` 8/16/32/64-bit natural numbers
`bigint` integers without upper or lower limit
`single_precision` single precision real number
`double_precision` double precision real number
`double_extended` double extended precision real number
`fixed_precision` fixed precision real number
`array` a mutable fixed-size sequence

The names for real numbers are borrowed from IEEE 754.

`list` a mutable variable-size sequence
`tuple` an immutable sequence
`char` a single character
`undefined` `undefined` in JavaScript
`null` `null` in JavaScript

The advanced data types are also where the language-specific aspects can come into play. For example, in addition to the basic type `nothing`, we have both `undefined` and `null`. In most languages, there is no difference between those, but for example, in JavaScript there is. Having both available as an advanced type allows exercises to use either in JavaScript exercises, while still being language-agnostic. Additionally, supporting both types allows for the creating of language-specific exercises for JavaScript, where one of the types is required but not

2.6.3. Statements and expressions

We return once more to the test suite from listing 2.1, which evaluates a function call. It illustrates the need for TESTed to support a language-agnostic description of expressions and statements. Evaluation of an expression yields a value, where TESTed allows describing expected values as identifiers (referring to a variable that was previously defined), function calls or literal values (using the data serialization format described in the previous section). A statement in TESTed is either an assignment (defining a new variable) or an expression (similarly to Python: every expression is also a statement in its own right).

Note the absence of control structures or operators in the description of expressions and statements. Our goal is not to create a full-fledged abstract programming language. We intentionally limit expressions and statements to features needed by TESTed. Creating a universal programming language that can be converted to all other programming languages is out of scope.

The use of assignments is illustrated by the expanded test suite in listing 2.2. We create a list and assign it to the variable `a_list`. Note that there are no explicit tests in this first test case. The test case will still fail if something unexpected happens, e.g. if an exception is thrown or output is written to standard error. The second test case then calls the function but uses the variable we defined as the first argument.

2.7. Evaluating submissions

This section discusses the complete process through which a submission is evaluated. The input for the evaluation process is a test suite and a submission, which is typically provided by the judge platform in which TESTed runs (or can be provided manually if TESTed is run on the command line). Subsequent subsections dive into more detail of individual parts of the process.

Figure 2.4 is a schematic overview of said evaluation process. Step zero in the evaluation process is checking if the exercise is solvable in the programming language of the submission (section 2.7.1). Next, the test suite is partitioned into compilation and execution units (section 2.7.2). For each of these units, the relevant test code is generated and compiled (section 2.7.3), in the programming language of the submission. This code generation is the bulk of language-specific code in TESTed, whose design and implementation are discussed in section 2.9.

After compilation, each resulting executable contains one or more execution units. These are then all executed, and the side effects (such as exceptions, standard output, standard error) and results (i.e. return values) are recorded (section 2.7.4). These results are then checked for correctness against the test suite (section 2.7.5). This results in the feedback, which is returned by TESTed.

TESTed also provides an opportunity for static analysis on the submission. Currently, all supported programming languages run a linter on the submission, the results of which are also included in the feedback as code annotations (section 2.7.6).

Examples include *ESLint*, *Pylint*, and *HLint*.

2.7.1. Correctness and solvability checks

The first step is doing some correctness checks. For example, the structure and contents of the test suite is checked with JSON Schema. TESTed takes a fail-fast approach (Shore 2004): instead of silently failing when a test suite contains an error, TESTed will always abort the evaluation with an error message. Additionally, all statements and expressions in the test suite are checked for syntax errors. Special care has been taken to provide useful error messages to the exercise author.

The next step in the evaluation process is checking if the test suite is usable for the programming language of the submission. This might not be the case for a number of reasons, the three main ones being:

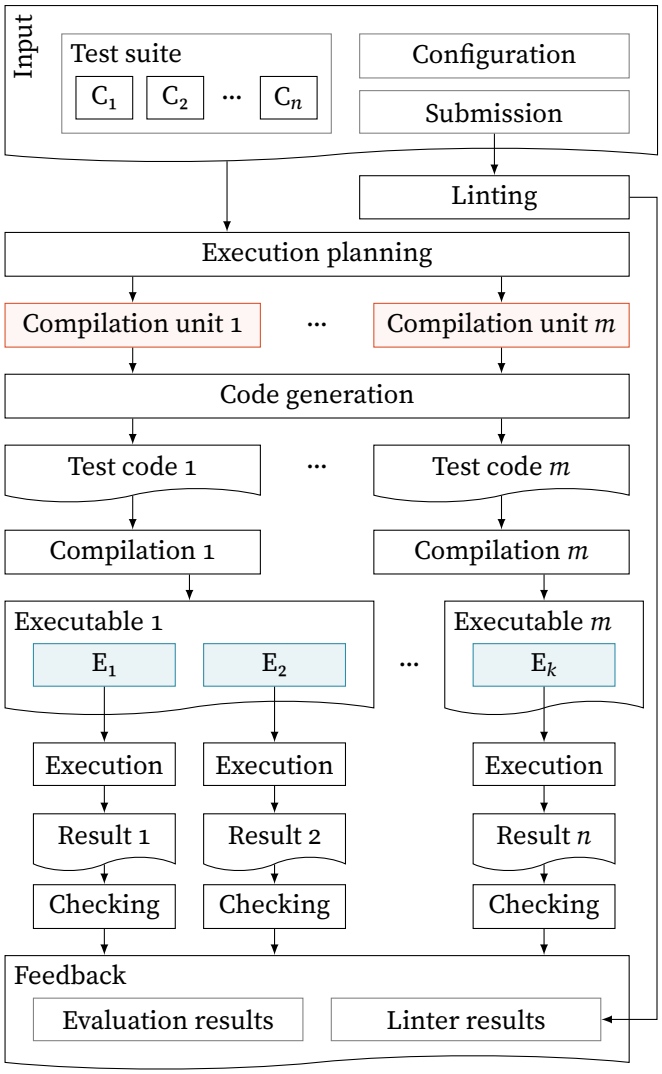


Figure 2.4. The process of evaluating a submission in TESTed. TESTed is typically run within a judge platform, which provides the submission, the test suite and the configuration options. The contexts in the test suite are organized into compilation units and then execution units. For each compilation unit, the test code is generated, which is then compiled. The execution units are then executed, which yields execution results. These are then checked, resulting in feedback. Separately, a linter performs static analysis on the code quality of the submission, and the resulting output is also included in the feedback.

- The exercise author has explicitly limited in which programming languages the exercise may be solved.
- The test suite uses constructs that are not supported by the programming language of the solution. For example, if the test suite uses object-oriented programming, the exercise will not be solvable in C or Haskell.
- The test suite contains programming-language-specific code but does not provide code for the programming language of the submission. For example, if a language-specific expression is only provided for Python, the exercise will only be solvable in Python.

2.7.2. Execution planning

The next step is planning the execution of the evaluation: creating an execution plan. As discussed before, a test suite contains a number of contexts, which must be independent of each other. TESTed partitions these into **compilation units** (a set of contexts that are compiled together), which are in turn partitioned into **execution units** (a set of contexts that are executed together). This partitioning is what we consider the execution plan.

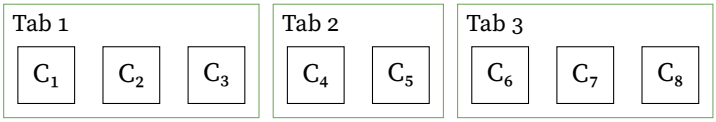
If performance was not a consideration, the simplest execution plan would be to compile and execute every context individually. After all, contexts are independent of each other, and separate compilation and execution would enforce that independence. However, this would be prohibitively slow: a test suite with fifty contexts would need fifty compilation steps and fifty execution steps. Creating an execution plan is thus necessary to improve performance, which is achieved by reducing the number of compilation units and execution units.

Compilation units

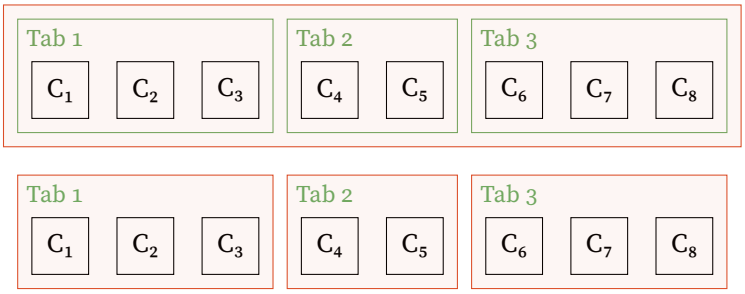
First, TESTed tries to use a single compilation unit for the whole test suite. This is achieved by creating a program that accepts an argument to indicate which execution should be run. In programming languages without an explicit compilation step, the compilation is no more than a syntax check. In compiled languages, the compilation is often much stricter, for example, failing if a non-existing function is used.

For example, our JavaScript implementation uses `node -c`.

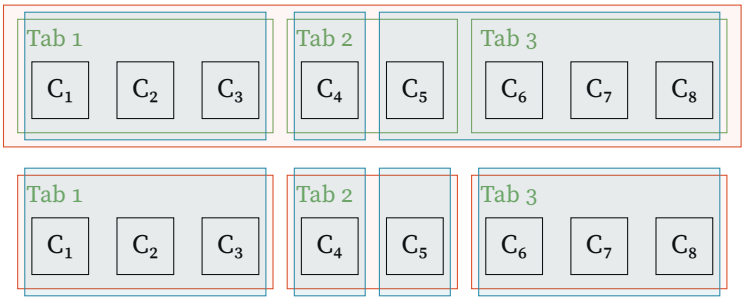
Techniques used to improve performance must be weighed against the usefulness of the generated feedback. For example, consider an exercise



(a) A schematic representation of a test suite, with three tabs and eight contexts. The tabs are represented with green boxes, while the contexts (denoted as C_n) are black boxes.



(b) The two possible partitionings for compilation units (which are denoted by red boxes). The upper partitioning consists of a single compilation unit for the whole test suite. This is always tried first. If this fails, each tab becomes its own compilation unit (the red boxes thus overlap with the green ones).



(c) Two possible partitionings for execution units (denoted by blue boxes), depending on the partitioning of the compilation units. In the first partitioning (top), the single compilation unit is split into three execution units. The second partitioning (bottom) cannot use the same execution units, as an execution unit cannot comprise multiple compilation units. The compilation units are thus further divided into more execution units.

Figure 2.5. Schematic representation of the planning steps.

where students must implement two functions (functions A and B). Students might submit a solution in which they only implement function A. With a single compilation unit, the generated test code contains calls to both function A and B. This will this result in a compilation error, as a call to a non-existing function is not allowed. This will prevent function A from being evaluated, even if it was correct.

To prevent this, if the compilation of the whole test suite fails, TESTed falls back to using one compilation unit per tab in the test suite. These two approaches are illustrated in figure 2.5b. This is why a tab is intended to be a set of logically related contexts. Continuing with the example from before, there might be a separate tab A for tests involving function A and a second tab for tests involving function B. Going more fine-grained, by compiling each context individually, does not seem useful. While potentially providing better feedback, the performance impact is significant. Compiling at the tab level is a good compromise between fine-grained compilation (thus allowing more of the submission to be evaluated) and performance (the more compilation units, the slower the evaluation will be).

Execution units

Next, the compilation units from the previous steps are partitioned into execution units. Each execution unit can be at most one compilation unit: we cannot execute multiple compilation units together, as each compilation unit results in a separate executable. However, depending on the contexts inside a compilation unit, we can (and do) execute a compilation unit multiple times.

All programming languages, including the non-compiled ones, are handled the same way. This allows the most consistency between programming languages.

For performance reasons, the ideal partitioning would be a single execution unit for the whole test suite. However, this prevents certain types of exercises from being evaluated correctly. Therefore, a new execution unit is started based on the type of context: if a context has standard input, command line arguments, or an explicit check for the exit code, a new execution unit is started.

2.7.3. Generating test code

Depending on the execution plan, the appropriate code is generated for the test suite. In concept, this step converts the language-agnostic test suite into actual source code, in the programming language of the test suite (an example of this is shown in section 2.4). For example, a submission in JavaScript requires generating code in JavaScript.

Code is generated for each compilation unit. The main purpose of the generated code is to execute the tests specified in the test suite. For this reason, we also call the generated code the **test code**. Besides executing the tests from the test suite, the generated code also contains some ceremonial code required by TESTed. Some examples of this ceremonial code are:

- As a compilation unit can contain multiple execution units, we generate a wrapper that executes the correct execution unit based on some parameter (e.g. `./testcode "unit1"`).
- The generated code includes serialization capabilities (section 2.6.2), to convert captured values into the internal data format used by TESTed.
- The tests are wrapped in code that captures side effects and values, such as exceptions, return values, etc.

This code generation is the main task of the programming-language-specific modules, discussed in section 2.9.

2.7.4. Executing test code

After the test code has been generated and compiled into executables, these executables are then executed.

During this execution, special care is taken to ensure that the executions are independent of each other. All execution takes place in a special directory called the *workdir* (from working directory), whose location is provided to TESTed via the configuration (section 2.8). For each execution unit, TESTed creates a new subdirectory and copies the relevant files into that subdirectory. The execution then happens in the subdirectory.

Since an execution unit consists of one or more independent contexts, they are also independent of each other. Another performance benefit of this independence is that this allows parallel execution of the different execution units. Of course, this is only relevant if there are multiple execution units, but this is the case with most exercises in our experience.

Another benefit is better feedback. Since TESTed copies all files into a dedicated subdirectory for each execution unit, we reduce chances of executions interfering with each other. For example, consider an erroneous submission for an exercise where data is provided in a file. If the submission overwrites or changes the file by accident, subsequent executions would use the modified file.

The astute reader might wonder how parallel execution is implemented, since TESTed is written in Python, a language whose default implementation (*CPython*) has an infamous global interpreter lock (GIL). This means that at any one time, only one thread can execute Python code. However, TESTed executes the test code in a separate process. This means the global interpreter lock does not apply: during execution of the test code, most of the time is spent waiting on results from the subprocess that is actually doing the execution. Waiting on input/output results is one area where the global interpreter lock is released, even in current Python versions.

Work is ongoing to remove the GIL from *CPython*, as PEP 703.

2.7.5. Checking test results

The code responsible for checking whether test results (return values, standard output, etc.) are correct is called an oracle (Howden 1978). TESTed supports three types of oracles, which can be divided into two categories: language-agnostic oracles that work for all programming languages and language-specific oracles, where each programming language requires a separate implementation of the oracle. Language-agnostic oracles are further divided into built-in oracles and programmed oracles.

The built-in oracles are provided by and included in TESTed. These are rather simple, but should nevertheless suffice for most exercises. We envision that as a natural evolution of using TESTed more broadly, other generic comparison methods might present themselves for inclusion in the framework. Currently, the following oracles are included:

Text oracle This oracle compares two strings and is used for standard output and standard error. The oracle has some options, for example, to ignore trailing whitespace, to attempt to parse the text as floating point numbers or case sensitivity. The expected value can be provided either as a string, or as a file, in which case the contents of said file are used as the string.

File oracle This allows comparison between two files, either comparing the whole file, or comparing the file line-by-line. When comparing line-by-line, the text oracle is used, so the same options can be provided.

Return value oracle This oracle compares two values (using the serialization from section 2.6.2). The oracle uses the basic and advanced types of TESTed to compare the data types in addition to the values.

Exception oracle This allows checking exceptions. By default, only the message of the exception is checked, as the type of an exception is programming language dependent. However, the oracle does have an option to provide the expected type for the different programming languages. Do note that checking the type happens with a string-based check, meaning that if a student implements a custom exception with the same name, it will pass the check.

There are a number of scenarios where the built-in oracles are not enough to properly check an exercise. One such example is an exercise where dynamic data is used, for example a return value that depends on the current date. Another example is a random or otherwise non-deterministic return value, where the value must satisfy some conditions. Finally, sometimes the exercise is static, but it is better to provide some exercise-specific feedback. For example, if the exercise is to generate an SVG, it would be better to include a rendered version of the SVG in the feedback.

Earlier versions of TESTed allowed implementing programmed oracles in any of the supported programming languages. Because of serious performance penalties, this is no longer allowed.

For these scenarios, a programmed oracle can be used. Such an oracle is, in essence, an extension of the built-in oracles: instead of using a built-in oracle, TESTed will call the exercise-provided oracle. Exercise authors must provide the oracle in the form of a function (the check function) implemented in Python. This oracle is also language-agnostic: the same check function will be used for all programming languages.

Lastly, there are the language-specific oracles, which are provided as an escape hatch. These oracles are run together with the generated test code and skip the data serialization pipeline. They are therefore not subject to the serialization limits of TESTed. An example of a scenario where this could be useful is an exercise where a function returns an instance of a custom class or any built-in datatype that does not map to a TESTed datatype.

The big downside to using these oracles is that they are programming language specific. The oracle thus has to be implemented in each programming language the exercise supports. These oracles are therefore only provided as an escape route for language-specific expressions, statements, and data types. We discourage their usage as much as possible.

2.7.6. Static analysis of the submission

TESTed also provides a way to perform exercise-independent static analysis on the submission as part of the evaluation process. The use of linters is widespread in the software engineering world and can also

benefit students by pointing to common errors and familiarize students with the programming style of the language they are working in (AlOmar et al. 2023). Currently, all supported languages (except C#, where the compiler acts a linter) in TESTed use an external linter to generate additional feedback (table 2.2).

Table 2.2. Overview of the linters used by TESTed. For C#, the compiler already provides linter-style warnings and feedback.

Programming language	Linter
Bash	Shellcheck
C	Cppcheck
C#	(compiler)
Haskell	HLint
Java	Checkstyle
JavaScript	ESLint
Kotlin	Ktlint
Python	Pylint

2.8. Integration with and influences of Dodona

Dodona (Van Petegem, Maertens et al. 2023) is an online platform for solving and submitting programming exercises. An introduction and overview of the platform is given in section 1.2. Relevant here is the distinction between a judge platform and testing framework (section 2.2): Dodona is the judge platform in which different testing frameworks (the judges in Dodona terminology) can be run.

While TESTed is available as a standalone tool, it was primarily developed with Dodona in mind. Therefore, some design decisions made by the Dodona platform still apply to TESTed as well. In this section, we briefly discuss the relevant aspects of Dodona that have an impact on TESTed. We also look at Dodona's feedback format, which is also used by TESTed.

2.8.1. Architecture of the Dodona platform

Dodona enforces a complete separation of the platform code and the testing framework. Communication between the platform and the testing frameworks is done via a well-defined interface, consisting of two parts: the input for testing frameworks and the feedback generated by testing

```
{
  // The programming language of the submission.
  "programming_language": "python",
  // The natural language used when submitting.
  "natural_language": "en",
  // Path to the resource folder of the exercise.
  "resources": "/exercise/simple-example/",
  // Path to the submission's source code.
  "source": "/exercise/simple-example/correct.py",
  // Path to the judge.
  "judge": "/tested/",
  // Path the a workdir, where execution should happen.
  "workdir": "/temp/workdir/",
  // Memory limit, in bytes.
  "memory_limit": 536870912,
  // Time limit, in seconds.
  "time_limit": 60,
  // Name of the test suite.
  "suite": "suite.yaml",
  // \textsc{test}ed: additional options...
  "linter": true
}
```

Listing 2.3. Annotated example of the input provided to testing frameworks by Dodona. This is also the input expected by TESTed.

frameworks. Both are discussed in the next two sections. Evaluating submissions is done using *Docker*, as testing frameworks run student code. They must be immune to bad code, e.g. a submission with an infinite loop should not bring down the platform. Neither should malevolent submissions: a fork bomb should similarly have no impact on the platform.

When code is submitted, the platform will create a Docker container using the testing framework's associated Docker image. Then, relevant files for the exercise and the submission are mounted into the container's file system. Finally, the container is run, which will start the testing framework inside the container with relevant options (section 2.8.2). Dodona then reads the feedback from the standard output of the container (section 2.8.3). This feedback is then saved into the database and shown to the user who submitted the code.

2.8.2. Dodona-provided input for testing frameworks

When Dodona starts a testing framework, it provides a JSON object on standard input. This configuration object contains all information

needed by the testing framework to evaluate a submission. An annotated example is provided in listing 2.3.

Most of the options are pretty straightforward. Of the generic options, the memory and time limit are informational: they are provided so that a testing framework can make an effort to limit submissions. However, Dodona will enforce these limits if the limits are exceeded. A common use-case is to provide better or more detailed feedback about the issue (since Dodona, by design, can only indicate a global memory or time limit exceeded error).

Additional testing-framework-specific options are also added to this object. For example, with TESTed, the name of the test suite is often such an option. Other usable options for TESTed[1] are:

`parallel` If contexts should be executed in parallel or not (default false). See section 2.7.4 for more information on this option.

`allow_fallback` Determines if unit compilation should be attempted if the global compilation fails (default true). See section 2.7.2 for more information about this option.

`linter` Enables or disables linting of the submissions (default true).

`language` An object mapping programming languages to objects containing language-specific options. As an example use-case of this option, some languages allow customizing the linter configuration.[2]

`compiler_optimizations` If compiler optimizations should be enabled if available or not (default false). By enabling this option, compile speed is sacrificed for better execution speed of the submission. This option can be useful for exercises where the submission is expected to be computationally heavy.

2.8.3. The Dodona feedback format

Dodona supports two output formats: a *full* and a *partial* output format. TESTed uses the partial output format, so we will only discuss that format.

The feedback has a similar structure as the test suite (section 2.6.1). The format is named partial since it is a streaming JSON format. This

[1] These are also described in our documentation at `https://docs.dodona.be/en/references/tested/exercise-config/`

[2] `https://docs.dodona.be/en/references/tested/exercise-config/#linters`

```
{"command": "start-judgement"}
{"command": "start-tab", "title": "Feedback"}
{"command": "start-context"}
{"command": "start-testcase", "description": "a_list = [1, 2, 3, 2]"}
{"command": "close-testcase"}
{"command": "start-testcase", "remove_all_occurrences(a_list, 2)"}
{"command": "start-test", "expected": "[1, 3]", "channel": "return"}
{"command": "close-test", "generated": "[1, 3]", "status": "correct"}
{"command": "close-testcase"}
{"command": "close-context"}
{"command": "start-context"}
{"command": "start-testcase", "description": "a_list = [0, 1, 1, 2]"}
{"command": "close-testcase"}
{"command": "start-testcase", "remove_all_occurrences(a_list, 1)"}
{"command": "start-test", "expected": "[0, 2]", "channel": "return"}
{"command": "close-test", "generated": "[0, 2]", "status": "correct"}
{"command": "close-testcase"}
{"command": "close-context"}
{"command": "close-tab"}
{"command": "close-judgement"}
```

Listing 2.4. Example of the output generated by TESTed, which is rendered in figure 2.3. As before, each context consists of two test cases, the first of which has no explicit tests, while the second has one test (the expected return value).

means that the output is a stream of JSON objects, instead of one big JSON object.

TESTed uses newline-delimited JSON. Two equivalent specifications exist for this format: NDJSON (Newline-Delimited JSON)[3] and JSON Lines[4]. The format itself is simple: JSON objects are separated by a newline, and each line is a valid JSON object.

Listing 2.4 contains the output from TESTed that resulted in the feedback as shown in figure 2.3. The structure of the feedback is indicated by commands, with `start` commands to begin a new level in the hierarchy and `close` commands to finish a level.

The Dodona feedback format is a simple, yet flexible format. It has been used by a variety of testing frameworks for general purpose programming languages (like TESTed, but also dedicated frameworks for JavaScript, Bash, Python, C, C#, Prolog, Haskell, R, Scheme, and assembly). It has also been used successfully for more niche testing frameworks (such as HTML/CSS, SQL, and Turtle).

This illustrates that TESTed is neither limited by this choice of output

[3] https://ndjson.org/
[4] https://jsonlines.org/

format, nor would it be challenging to support this format in other platforms.

2.9. Programming language support

The parts of the evaluation process that are programming-language-specific are implemented using a module system. To make adding new programming languages easy, TESTed enforces a strict separation of concerns with regard to language-specific tasks. All language-specific actions and tasks must go through a single well-defined interface. This interface is implemented using Python's object-oriented capabilities by defining an abstract base class called `Language`.

This `Language` class has a set of abstract methods, for which an implementation is necessary, and a set of optional methods, which may be overridden but are not required. The `Language` class is the interface between the core modules of TESTed and the language-specific modules. No other modules have language-specific code. The main task when adding support for a new programming language is to implement this abstract base class. Some other smaller tasks are registering the language in TESTed and adding support in the test suite for this new programming language.

The remainder of this section describes the different methods that must or can be implemented. We always begin by providing the method signature, followed by a discussion of the method. Most of this information is also available in the class itself as documentation in the code, which is also the most up-to-date version.[5]

2.9.1. Compilation

```
1 def compilation(self, files: list[str]) -> CallbackResult:
```

The compilation step of the evaluation process is responsible for compiling the generated test code (with the compilation units) into executable (section 2.7.2). This method implements this step and must return the command that TESTed will use to compile the compilation unit. The return type `CallbackResult` is an alias for `tuple[Command, list[str] | FileFilter]`.

[5] https://github.com/dodona-edu/universal-judge/blob/master/tested/languages/config.py

The only argument (`files`) of this method is a list of files that the compilation unit comprises of. By convention, the file with the *program entry point* (which is often a main function) is last in the list.

The first value of the returned tuple is the compilation command. This command is a list of strings, which will be executed with the Python `subprocess` package. The second part of the return value must be a list of generated files, in which by the same convention the last file is the executable file. All files in this list will be made available to the execution command in the next step of the evaluation process. Alternatively, a file filter can be returned, which allows dynamic filtering of the resulting files after compilation.

As an example, consider the C language. When compiling C, we are only interested in the resulting binary, which also has a predictable name. The list of generated files can thus contain a single string: the name of the generated binary.

However, it is not always possible to predict the list of generated files, nor is it possible to predict their names. For example, in Java, compiling a file will result in one or more class files, depending on the content of the Java file (a nested class will result in more class files). In that case, the file filter can be used, which will be called for each file in the compilation directory after compilation has completed.

As a concrete example, this is how the method is called and what its return value is for C (on Windows):

```
>>> compilation(['submission.c', 'evaluation_result.c',
↪  'context_0_0.c', 'selector.c'])
(
    ['gcc', '-std=c11', '-Wall', 'evaluation_result.c', 'values.c',
    ↪  'selector.c',
     '-o', 'selector.exe'], ['selector.exe']
)
```

The compilation method is optional: languages that do not require compilation can use the default implementation. However, this step is ideal to at least perform a syntax check, and we recommend that all languages do this, if at all possible. Even non-compiled languages often have a syntax checker that is faster than executing the program. For example, both Python and JavaScript are not considered compiled languages, but both implement a syntax check in TESTed.

2.9.2. Execution

One of the most important methods is the method responsible for creating the execution command. This method is called after the compilation step, if that step was successful.

```
def execution(self, cwd: Path, file: str, arguments: list[str]) ->
↪  Command:
```

This method must return one value: the command to execute. As with the compilation method, the returned command will be executed by passing it to Python's `subprocess` package.

The returned command must execute the file from the `file` argument. The argument `arguments` contains a list of command line arguments that must be passed to the program. The `cwd` arguments is the directory in which the execution will take place. This can be useful for languages that compile to a binary executable. Since this executable is not on the path, it is safer to return an absolute path to it.

Continuing with the same example in C, a call to this method would look like this:

```
>>> execution('/test/path', 'executable.exe', ['arg1', 'arg2'])
['/test/path/executable.exe', 'arg1', 'arg2']
```

All files that were included in the return value of the compilation method will also be available in the execution directory, in addition to other dependencies that we discuss next.

2.9.3. Dependencies and other files

In the commands from the two previous sections, the methods receive a list of files that are potential dependencies for compilation or execution. There are also some methods that optionally can influence which files are considered dependencies.

```
def initial_dependencies(self) -> list[str]:
```

Returns a list of additional dependencies that will be included in the compilation and execution. The returned strings are paths to files, relative to the implementation folder of the language module in TESTed.

For example, most languages include a separate file to deal with encoding return values into the TESTed data format.

```
def filter_dependencies(self, files: list[Path], context_name: str)
↪  -> list[Path]:
```

Used to filter the results of the compilation step to the files needed for one test case. In most cases, a single compilation step is used for all test case. However, not all languages need all resulting files for each execution. By default, the name of the test case is used to filter the files.

```
def find_main_file(self, files: list[Path], name: str) -> Path |
↪  Status
```

This optional method finds the main file (meaning the executable file or the file with the main method) in a list of dependencies. The method should either return the path to the main file, or return an error status if the file could not be found.

```
def modify_solution(self, solution: Path):
```

A callback that allows modifying the submission. The submission should be modified in place. The callback is called after linting, but before compilation or execution.

An example of this use case is JavaScript. To support both CommonJS and ES6 modules, we analyse the code and add exports for all functions, variables, and classes in the submission. Similarly, the main function in C programs is renamed to prevent conflicts with the main function in the generated TESTed code.

2.9.4. Configuration and conventions

There are a number of simple methods that deal with the different conventions used by the programming language.

```
def get_string_quote(self) -> str:
```

Returns the character used to quote strings. By default, this is a double quotation mark (").

```
def naming_conventions(self) -> dict[Conventionable,
↪  NamingConventions]:
```

Returns the naming conventions used by the programming language. It must return a dictionary, which maps different aspects to a naming convention.

The “conventionable” aspects are namespaces, function names, identifiers, properties, classes, and global identifiers. Most are self-explanatory, except for namespace, whose meaning depends on the programming language. In some languages this is used as the name for packages or modules, but in Bash, for example, it is used as the name of the script.

Table 2.3 contains an overview of the available naming conventions in TESTed.

In practice, most languages use camel case and snake case (and their uppercase variants).

Table 2.3. Available naming conventions in TESTed.

Naming convention	Example
Camel case	`thisIsAnExample`
Snake case	`this_is_an_example`
Camel snake case	`this_Is_An_Example`
Cobol case	`THIS-IS-AN-EXAMPLE`
Dash case	`this-is-an-example`
Donor case	`this|is|an|example`
Flat case	`thisisanexample`
Macro case	`THIS_IS_AN_EXAMPLE`
Pascal case	`ThisIsAnExample`
Pascal snake case	`This_Is_An_Example`
Train case	`This-Is-An-Example`
Upper (flat) case	`THISISANEXAMPLE`

```
def file_extension(self) -> str:
```

Return the main file extension for the programming language. For a language with multiple extensions, this should return the extension used for executable (main) files. For example, in C this returns `c` and not `h`.

```
def is_source_file(self, file: Path) -> bool:
```

An optional method that determines if a file could be a source file for the programming language. By default, this will check the extension of the file against the extension provided by the `file_extension` method.

```
def submission_file(self) -> str:
```

Returns the name of the submission file. By default, this calls the helper function `submission_name` and adds the file extension to it.

2.9.5. Type support

The system for data types used by TESTed is explained in section 2.6.2. The actual implementation of this system happens with the following methods.

```
def supported_constructs(self) -> set[Construct]:
```

This method should return a set of the constructs that are supported by this language. This is one of the mechanisms used in section 2.7.1 to check if a submissions in a certain programming language are possible for a given test suite.

The currently supported constructs are:

Objects Object-oriented constructs, such as classes.
Exceptions Exception support.
Function calls Function call support.
Assignments The result of an expression can be assigned to a variable.
Heterogeneous collections Data structures whose elements can be of different data types.
Default parameters Parameters in a function with a default value.
Named parameters Parameters in a function can be passed by name rather than (or in addition to) by position.
Global variables Variables or constants defined at a top-level.

Note that the constructs can sometimes be interpreted in a loose sense. For example, the Haskell implementation indicates that assignments are supported, even if this is not strictly true. `x = 5 + 5` defines a new function `x` with the body `5 + 5`. However, for practical purposes, this can fulfil the same role as an assignment in TESTed.

```
def collection_restrictions(self) -> dict[AllTypes,
↪  set[ExpressionTypes]]:
```

This optional method allows restricting which data types are allowed in collection types. Currently, restrictions for map keys and sets are supported. For example, in Python, a list is not hashable, meaning it cannot be used as the key in a dictionary, nor can it be an element of a set.

```
def datatype_support(self) -> dict[AllTypes, TypeSupport]:
```

This function is used to indicate the data type support for the language. The return value is a mapping of the types to their support. The default is unsupported: only supported types must be present.

For example, in Bash, the implementation of this function looks like:

```
def datatype_support(self) -> dict[AllTypes, TypeSupport]:
    return {
        AdvancedStringTypes.CHAR: TypeSupport.REDUCED,
        BasicStringTypes.TEXT: TypeSupport.SUPPORTED,
    }
```

Strings are supported (and are the only type supported by TESTed). Characters are supported, but in reduced form. This means that strings will be used for characters.

2.9.6. Stacktraces and compiler outputs

```
def cleanup_stacktrace(self, stacktrace: str) -> str:
```

In most cases, stacktraces from runtime errors, compiler errors, or compiler warnings contain references to the generated TESTed code. However, these lines are not relevant nor useful to students. Therefore, this method provides a way to clean up stacktraces.

```
def compiler_output(self, stdout: str, stderr: str) ->
↪  tuple[list[Message], list[AnnotateCode], str, str]
```

Another example of a use-case is adding links in the feedback to the relevant lines in the submission. Dodona also supports this feature, so users there can click on a stacktrace and go to the relevant lines, similar to most development environments. The returned tuple contains a list of messages, a list of code annotations, and the clean version of the compiler output.

2.9.7. Code generation

TESTed works by generating code (section 2.7.3). The following methods are called on the language module to generate code for various language constructs. In the implementation for most languages, the code generation is implemented in a separate module, and these methods just call that module.

```
def generate_statement(self, statement: Statement) -> str
```

Generate code for a statement (since a statement is also an expression, this also covers values).

```
def generate_execution_unit(self, execution_unit:
↪  "PreparedExecutionUnit") -> str:
```

Generate code for an execution unit. It is expected that the implementation of this method uses the other methods for generating code. For example, an execution unit probably needs to generate code for a statement somewhere, which would be done using `generate_statement`.

When generating the code for an execution unit, a few things must be taken into account (figure 2.6):

- TESTed expects two sentinel values to be present in all outputs (standard output, standard error, return values, exceptions): a secret value outputted between the outputs for test cases and another secret value between the outputs for contexts.

- Return values and exceptions must be serialized in JSON, using the internal data form from TESTed.
- The generated code should be robust against unexpected output, including exceptions.

2.10. Evaluation of the TESTed framework

To validate whether TESTed meets the requirements put forward in section 2.3, we report on its use in educational practice. We conducted three quasi-experiments (table 2.4) where we asked students to solve a set of programming exercises and automatically reviewed their submissions with TESTed in Dodona. Each experiment specifically focuses on a particular aspect of TESTed we want to evaluate.

Table 2.4. A summary of the three quasi-experiments by their data.

Experiment	Languages	Users	Exercises	Submissions
Language independence	6	38	3	468
Overhead for designers	6	325	50	5465
Educational practice	1	95	5	6696

2.10.1. Programming language independence

The goal of a programming-language-agnostic testing framework is to allow unit testing with a programming-language-agnostic test suite. We therefore wanted to verify that this goal can be achieved with a testing framework that implements the formulated requirements. We asked higher education students to solve a set of three programming exercises in one or more programming languages supported by TESTed at that time (C, Haskell, Java, JavaScript, Kotlin, Python). One exercise required solutions to read data from standard output and generate results on standard output, i.e. the classic ACM-ICPC style programming challenges. The other exercises required implementing one or more functions that are called many times from the test suite with different arguments to cover different corner cases.

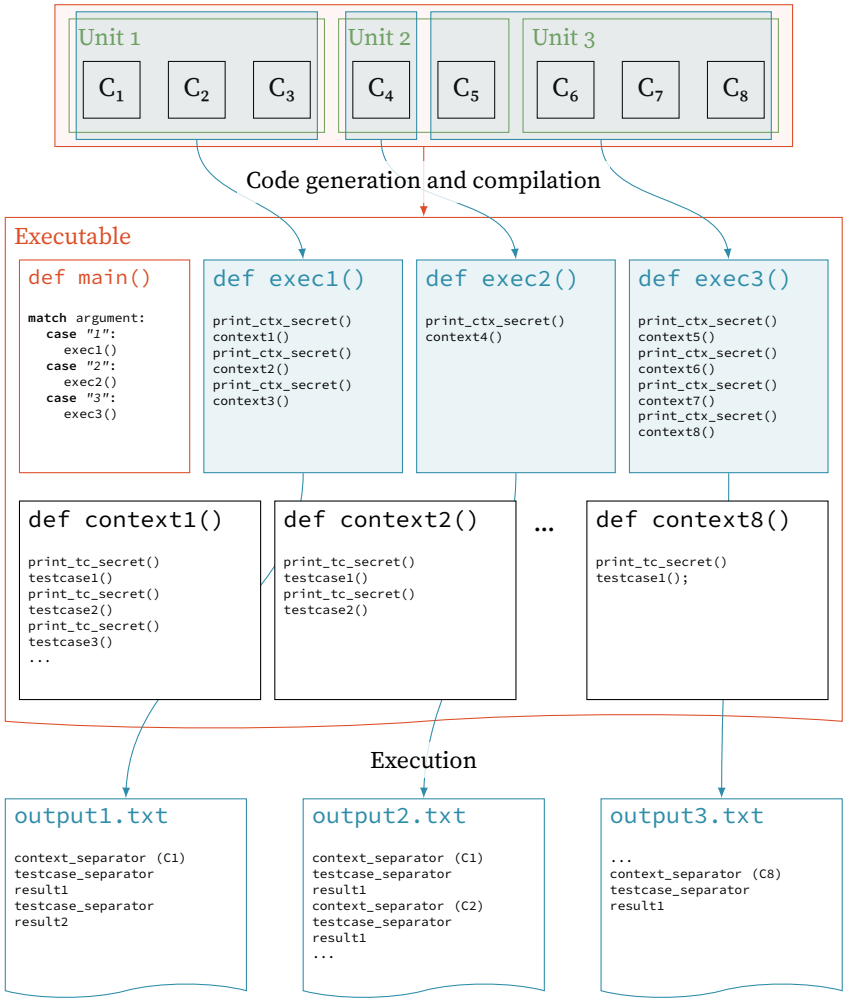


Figure 2.6. Overview of code generation in TESTed. This figure covers the optimal scenario, where a single compilation unit for the whole test suite is used (figure 2.5b). Its main function has one job: selecting the correct function for the relevant execution unit. This is illustrated here with a `case`. Each of these execution functions will write a secret (to separate the outputs) to all outputs (return values, standard output, standard error, exceptions) and will then call the function for the contexts in that execution unit. This function for the context does something similar: it writes the context secret to the outputs and execute the test cases. The final results for each output are a set of captured values separated by the context and test case separators.

We challenged students to solve the exercises in as many programming languages as they possibly could, in order to broadly cover all programming languages supported by TESTed. We also added support for new programming languages to TESTed while students were solving the exercises, in order to verify that test suites were robust against these extensions. TESTed automatically evaluated submissions in different programming languages for all three exercises immediately upon submission. Afterwards we manually verified that valid feedback was provided for all supported languages.

In total, 38 users (including several of the authors of Strijbol, Van Petegem et al. (2023)) submitted 468 solutions for the three exercises. Upon evaluation by TESTed based on the test suites for the exercises, 158 submissions passed all test cases and 310 submissions did not compile or failed at least one test case. 44 (28%) of the correct submissions were implemented in Python, 28 (17%) in Haskell, 27 (17%) in JavaScript, 22 (14%) in C, 19 (12%) in Java and 17 (11%) in Kotlin.

As expected, TESTed indicated that it could not evaluate submissions in C for one of the three exercises. The test suite for this exercise specified that some function calls needed to return an array and this is not supported by the configuration of the C programming language in TESTed. Support for arrays in C is omitted intentionally, as C has no first-class array support. For example, to pass an array as a function parameter in C, it is common to pass a pointer and the length of the array. Similarly, returning a pointer from a function is not enough; the length also needs to be passed somehow. One solution is to return a `struct` containing both the pointer to the array and the length, but this is not standard in C.

We consider the experiment successful. Apart from the arrays in C, no other limitations of TESTed were encountered. All submissions that were considered correct upon manual inspection also passed all test cases upon evaluation by TESTed, irrespective of the programming language. All submissions that were considered wrong upon manual inspection failed to pass the evaluation by TESTed either at compile time or when executing the test cases. Additionally, we received no complaints by students of programming style issues. For example, TESTed correctly converted between camel case and snake case. Another example is JavaScript, where TESTed supports both synchronous and asynchronous functions.

2.10.2. Overhead for exercise authors

The main goal of the second experiment was to identify any overhead that could possibly be imposed on exercise authors while producing test suites for TESTed, compared to authoring test suites for a specific programming language. A secondary goal was to identify possible limitations on the kind of exercises that TESTed could evaluate automatically or any shortcomings in the feedback it provides. For these purposes we designed test suites for all programming challenges of the 2020 edition of the Advent of Code, an annual programming contest run by Eric Wastl from December 1 to December 25.[6]

The Advent of Code platform does not directly review source code that solves the programming challenges, but provides users with a single input data set and accepts a single textual result (usually a number or a short text) that is compared against the expected result by exact string matching. We have designed more elaborate test suites for all Advent of Code challenges that allow for more fine-grained testing of implemented solutions as is common practice in software engineering and in educational practice.

Following a divide and conquer strategy, most challenges were broken down into multiple functions or methods that compute intermediate results that can be tested explicitly. Each function and method was tested separately using multiple input data sets (50 test cases by default) that were either provided as arguments when calling the functions/methods or as a text file containing bulk data. The strongly typed features of TESTed were fully exploited in passing arguments (including file locations) and specifying expected return values. As a result, users got richer and more granular feedback from TESTed compared to the binary feedback (correct/wrong) from the Advent of Code platform, as test suites were also checking intermediate computations in addition to final results, and were testing on multiple input data sets with varying sizes and covering different corner cases.

In total, 325 users (including several of the authors of Strijbol, Van Petegem et al. (2023)) submitted 5465 solutions for the 50 challenges in the 2020 edition of the Advent of Code. Upon automatic review by TESTed, 1844 submissions passed all test cases and 3621 submissions did not compile or failed at least one test case. 1507 (81%) of the correct submissions were implemented in Python, 141 (8 %) in Java, 78 (4 %) in JavaScript, 57 (3 %) in C, 41 (2 %) in Haskell, and 20 (1 %) in Kotlin. Compared to the previous experiment, submissions for the Advent of Code challenges

[6] `https://adventofcode.com/`

are much more skewed towards Python. Most users participating in this experiment were students that were driven by solving the Advent of Code challenges on a daily basis using the programming language they were most familiar with or they wanted to practice more. Solving the challenges using multiple languages, as we did for the first experiment, was not explicitly promoted in this case.

In light of the goals put forward for this experiment, we again evaluated the experiment as successful. On a daily basis from December 1 to December 25, we could publish the two Advent of Code challenges on our learning platform with support for automatic review by TESTed within two hours after they had been published on the Advent of Code platform. This required a single test suite designer to implement a solution for an Advent of Code challenge, design an interface (functions or methods) for solving the challenge, generate 50 test cases that call each function/method with a diverse set of arguments (using the implemented solution to compute the expected return value). In addition, the same test suite designer also translated the description of the challenge in Dutch because our learning platform supports programming challenges both in Dutch and English. Meanwhile, some of the other authors (Strijbol, Van Petegem et al. 2023) were validating the test suite design and the feedback provided by TESTed using their own implementations of the Advent of Code challenge in a variation of programming languages.

We found that generating generic test suites for TESTed was not more time-consuming than generating test suites for the testing framework of a specific programming language. We have been able to design test suites for all Advent of Code challenges without discovering any limitations or needs to find workarounds. All test suites rely only on the built-in oracles of TESTed, so for the Advent of Code challenges we never had to rely on the programmed or language-specific oracles. The latter underscores that more advanced or language-specific techniques for automatic evaluations are indeed only needed in specific cases.

We could no longer verify the feedback generated by TESTed for all submitted solutions, but we did not encounter any invalid feedback when inspecting a sample of the submissions while running the challenges. Users could contact the authors (Strijbol, Van Petegem et al. 2023) through the Q&A module of Dodona and we explicitly asked them to report any bugs or shortcomings observed while submitting their solutions and receiving feedback, but all responses ($n = 30$ during the 2020 edition) were questions about how to solve some of the Advent of Code challenges or about suggestions on how to improve (the performance) of submitted solutions. No responses hinted at improvements we could make on TESTed.

This experiment also revealed a first area for future work. We wanted to explore how the process of designing programming exercises with automated feedback provided by TESTed can be made more ergonomic. While test suites expressed in JSON can be easily computer generated, their format is less suitable for reading and writing by humans. At the time of writing the publication of this chapter, we were already looking into designing a domain-specific language to describe test suites for programming exercises.

Our idea then was that TESTed would convert the domain-specific language into JSON as a preprocessing step, while keeping JSON as the base format for computer generated test suites. However, as detailed in chapter 3, we later used the domain-specific language directly, without conversion step. We also investigated the option to support multiple domain-specific languages for different types of programming exercises, for example a DSL for stdin/stdout exercises and another domain-specific language for test suites built around function calls. However, it became clear the supporting both scenarios in the same domain-specific language was feasible and a better approach. Also of interest was supporting previous attempts at standardizing descriptions of programming exercises such as YAPExIL (Paiva, Queirós et al. 2020) and PEML (Mishra and Edwards 2023), and extending BabelO (Queirós and Leal 2013) with support for TESTed.

Designing a programming exercise that can be solved in multiple programming languages not only requires a programming-language-agnostic testing framework, but also a task description that adapts itself to selected programming languages. Particular differences between languages that need to be taken into account in expressing task descriptions are naming conventions (e.g. camel case or snake case), names for data types (e.g. lists in Python or arrays in Java), representation of literals (e.g. single quotes or double quotes as string delimiters) and grammar for expressions and statements (e.g. in sample code snippets).

We also investigated the use of a templating system to describe task descriptions in a programming-language-agnostic way, in combination with TESTed as an engine to replace generic placeholders with language specific descriptions on the fly. In this context we looked into extending the restricted support for expressions and statements that TESTed uses to denote literals, variables, assignments, function calls and object creation in a generic way into a more expressive abstract language that, for example, also supports mathematical and logical operators. This was implemented with the DSL as well, as discussed in section 3.2.4.

2.10.3. TESTed in educational practice

Where previous experiments were primarily conducted with higher education students, the experiments themselves were run outside regular educational practice. Therefore, we ran a third experiment where we replaced the automatic evaluation using a testing framework for JavaScript with evaluations provided by TESTed for some of the exercises halfway through the semester of a course with higher education students. Students were not informed beforehand that the automated feedback would be provided by another testing framework. The existing test suite for the JavaScript testing framework was replaced with a test suite for TESTed, using exactly the same test cases, so that we could immediately switch back to the JavaScript testing framework in case any blocking issues would occur. We configured TESTed to evaluate all submissions as JavaScript code, so students could not solve exercises in any other programming language than JavaScript.

In total, 95 students submitted 6696 JavaScript solutions for five programming exercises. Upon automatic review by TESTed, 501 submissions passed all test cases and 6195 submissions did not compile or failed at least one test case. Note that Dodona (Van Petegem, Maertens et al. 2023) does not impose any restrictions on the number of submissions students can make for the programming exercises.

The students did not spontaneously send any signals that they noticed differences between the feedback they received from the JavaScript framework or TESTed. After we informed the students about the change we made in the background of Dodona, they confirmed that the feedback provided by TESTed was on par with feedback they received before. Some students had noticed the introduction of linting messages to their submissions, which TESTed provides for all supported programming languages (using *ESLint* for JavaScript). However they believed this was an addition to the existing JavaScript framework and identified it as an improvement on the feedback they received.

As an extension to the experiment, we prepared a JavaScript exercise that was automatically reviewed using both the JavaScript testing framework and TESTed. Students were asked to submit their solutions twice and compare the feedback they received from both frameworks. Except for the additional linter information provided by TESTed, students did not observe any significant differences in the quality of the feedback they received. However, some students did notice that it took somewhat longer to get feedback from TESTed than from the JavaScript testing framework.

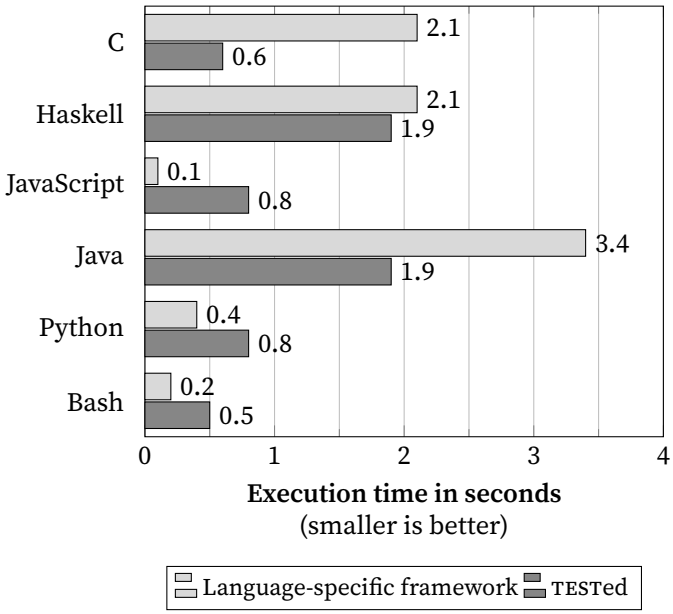


Figure 2.7. Runtime performance of TESTed versus language-specific testing frameworks. For each programming language, we evaluated a correct submission for the same exercise with a language-specific testing framework and with TESTed.

Prompted by this feedback, we investigated the performance of TESTed and compared it against the language-specific testing frameworks we routinely use in our educational practice. We measured the execution time of the evaluation of a correct submission for one exercise, using both TESTed and a language-specific testing framework (figure 2.7). This way we were able to compare timings for Bash, Python, Java, JavaScript, Haskell and C. We did not include Kotlin in this experiment, a language that is also supported in TESTed, as Dodona does not have a language-specific testing framework for Kotlin. The exercise used is simple: it requires implementing a function `echo` that returns its argument. As a result, the time measured is almost completely testing framework overhead, as the execution time of the submission is negligible. The implemented function was called 50 times with different arguments.

Our performance analysis indicates that an evaluation by TESTed is slower for Bash, Python and JavaScript. This does not come as a surprise, since TESTed needs to do additional work in generating test code in the programming language of the submission and in serializing the results after executing the test code. In addition, there are also differences in

what evaluations are performed. For example, the JavaScript judge does not lint submissions where TESTed systematically applies linting for all supported languages. However, evaluations by TESTed are faster for Java, Haskell and C. This is mainly due to implementation differences in the testing frameworks, especially in the compilation step. In TESTed, we have implemented the compilation step (and code generation) with as little overhead the case, which is not always possible in language-specific testing frameworks. For example, the Java framework relies on jUnit, while TESTed does not have any Java dependencies aside from the Java language itself.

Performance remains a focus area for future work. Automated testing frameworks often deliver just-in-time feedback and are integrated into highly interactive learning environments. As a result, students who frequently submit solutions during hands-on sessions or while working on homework assignments expect immediate results and get frustrated by poor response times. Performance is therefore critical in educational software testing. When implementing TESTed, we paid specific attention to reduce overhead during test code generation, compilation and execution, for example by bundling multiple contexts in a single compilation and execution step, making the performance of TESTed acceptable.

However, conditions in which TESTed may bundle contexts could still be improved, so that more contexts from the same test suite could be compiled and executed together. Since test code generation only depends on a test suite and a selected programming language, we might also consider caching as a way to reuse generated test code for all submissions of a programming exercise that share the same programming language. Performance could also be boosted by linting submissions in parallel to testing them, where TESTed currently runs these two steps sequentially.

2.11. Conclusion

Educational software testing is the application of testing frameworks to provide automated feedback on solutions that students submit for programming exercises. We identified input/output testing and unit testing as two opposing strategies commonly used in educational practice, and investigated the impact of both approaches on programming language support of the frameworks. Testing frameworks adopting unit testing enable fine-grained software testing, but are highly language specific. Input/output testing, on the other hand, is more generic in that it supports

multiple programming languages, but imposes severe restrictions on programming exercises, granularity of testing and quality of feedback.

Our goal was to combine the best of both worlds. We formulated requirements for programming-language-agnostic testing frameworks that combine unit testing with support for multiple programming languages. We see three clear benefits for the adoption of such frameworks. First, exercise designers only need to know and use a single testing framework to create programming exercises with support for automated feedback, irrespective of the target programming language, instead of switching to a new testing framework for each programming language, taking into account restrictions on what can be tested or giving up on the quality of feedback.

Second, programming exercises only need a single test suite to evaluate solutions in a multitude of programming languages. This allows teachers to reuse the same programming exercise for a language of their choice or to give students the freedom to solve exercises in their preferred language. Third, it also saves time and effort to support educational software testing for new programming languages as common functionality of testing frameworks is implemented once in a generic way, and only needs to be complemented with a thin layer of language specific configurations for each individual programming language.

To validate the feasibility of designing programming-language-agnostic testing frameworks, we implemented a proof-of-concept framework called TESTed. Having such a prototype also enabled us to evaluate the framework in educational practice. The realization of TESTed confirms that the requirements for programming-language-agnostic testing frameworks can be met and provides a framework that can be used in educational practice. At the same time, working on and with TESTed also brought forward some areas for improvement and further research.

Chapter 3.

A domain-specific language for creating programming exercises

> *Programs must be written for people to read, and only incidentally for machines to execute.*
> — Abelson & Sussman, *Structure and Interpretation of Computer Programs*

Automated software testing is widely used in programming education to validate the correct behaviour of submissions for programming exercises. There are two dominant testing approaches: input/output testing and unit testing. Input/output testing is largely independent of the programming language, but its black-box nature makes it difficult to provide detailed feedback. Unit testing, on the other hand, typically requires a separate test suite for each target programming language.

This chapter introduces TESTed-DSL as a domain-specific language (DSL) designed to simplify authoring language-agnostic test suites for programming exercises with automated assessment support. Test suites written using TESTed-DSL *i*) share the same declarative structure and testing functionality across programming languages, *ii*) bridge the gap between input/output testing and unit testing, and *iii*) allow for expressing test code in a language-agnostic way. The educational software testing framework TESTed allows for automated assessment using language-agnostic test suites expressed in TESTed-DSL, as demonstrated in a case study. Additionally, TESTed now includes a template engine based on TESTed-DSL for authoring task descriptions that include language-agnostic specifications of data types, literal values, identifiers, and code fragments.

3.1. Background and motivation

3.1.1. Educational software testing

Software testing is the validation and verification of a system derived from source code (Ammann and Offutt 2016). Two complementary approaches prevail: dynamic testing executes the source code with a given suite of test cases, whereas static testing analyses the source code without executing it (Romli et al. 2010). Both approaches might be done via manual or automated processes, based on a specification (Pieterse 2013). In modern software development, automated testing has become a standard practice for continuous integration and continuous development of living codebases, where both the codebase and the system requirements might evolve over time (Winters et al. 2020).

The minimum requirement that is tested is correctness, which is the essential purpose of software testing (Pan 1999). The desired or correct behaviours are specified as the functional requirements of the program and say how a program must behave (Bass et al. 2021). Automated testing for correctness needs some kind of oracle to tell if the functional requirements are satisfied. Other software quality factors (the non-functional requirements) that may be tested are its functionality (reliability, usability, integrity), engineering (efficiency, testability, documentation, structure) and adaptability (flexibility, reusability, maintainability) (Hetzel 1988).

Educational software testing is the application of automated software testing to source code students submit for programming exercises (de Souza et al. 2016; Keuning et al. 2018; Paiva, Leal et al. 2022; Staubitz, Teusner et al. 2017). For brevity, the source code under test is be called a submission. What makes educational software testing unique is that each programming exercise has a fixed specification, against which multiple submissions must be validated and verified (Wilcox 2016). Submissions are usually small to moderate in size, with all source code contained in a single file in most cases.

The main purpose of educational software testing is automated assessment of submissions (Berssanette and de Francisco 2021), which may come as feedback and/or as a grade (Caiza and del Alamo 2013). Assessments can be provided instantly upon each submission while students work on their solution. Such a continuous assessment provides "feed back" on how students performed and "feed forward" on what to do next before making a new submission for the same assignment (Cheang et al. 2003; Higgins et al. 2003; Luck and Joy 1999). Assessments can also happen after a submission deadline has passed, and provide students

"feed back" on their overall performance (Hattie and Timperley 2007; Timmis et al. 2016).

3.1.2. Programming exercises

Efforts to standardize the representation of programming exercises have been numerous, encompassing various strategies, such as combining the metadata, assets and task description into one formally structured document (Mishra and Edwards 2023; Paiva, Queirós et al. 2020; Queirós and Leal 2011, 2012; Swacha 2018), a predefined directory structure (Verhoeff 2008), or a combination thereof (Edwards et al. 2008; Strickroth et al. 2015). The objective of these initiatives is laudable: increasing the FAIRness of programming exercises as digital educational resources that are Findable, Accessible, Interoperable and Reusable (Wilkinson et al. 2016). Without exception, all of them support (automated) assessment, showing its importance for programming exercises. However, due to their generic nature, these exercise standards only provide support for assessment metadata and assets, and leave the actual representation of test suites, software testing frameworks, and dependencies on runtime environments open for implementation. But to make programming exercises truly interoperable, test suites and testing frameworks are no implementation detail. As a result, any promise of plug-and-play programming exercises that can be freely exchanged between learning platforms has not yet been resolved (Ala-Mutka 2005; Ihantola et al. 2010; Messer et al. 2024; Paiva, Leal et al. 2022).

In this chapter, we do not focus on the overall representation of programming exercises as such, but rather on the specific aspects of task descriptions and test suites, which are key for automated assessment. We explore how to design programming exercises that can be assessed automatically across different programming languages. This capability is essential for creating exercises that accept submissions in various languages and allow for dynamic testing using a single test suite (Staubitz, Klement et al. 2015). For that purpose, test suites capturing the requirements of programming exercises must ideally be specified in a language-agnostic way. We focus in the first place on validating the correctness of submissions as a way of formative feedback, and not per se on grading submissions as a way of summative feedback.

Designing programming exercises that apply across programming languages is relevant in educational practice (Murphy et al. 2017). Exercises whose primary focus is problem-solving, data structures or algorithms are by nature only loosely bound to a particular programming language,

so teachers may want to leave the choice of language to individual students. This is especially useful in courses taken by mixed populations of students trained in different programming languages or where choice of the most appropriate language is an explicit challenge in the problem-solving process. But even programming courses that teach a particular programming language might benefit from exercise repositories built around the FAIR-principles, where exercises restricted to a single programming language are simply less reusable.

3.1.3. Input/output testing

The need for programming exercises supporting automated assessment across programming languages is also reflected by the fact that the most often used architecture for test automation in educational practice is based on standard input and output (Douce et al. 2005; Ullah et al. 2018; Wasik et al. 2018). The only language-specific step with such a data-driven approach is the optional compilation and then execution of submissions, which read from the standard input stream (stdin) and write to the standard output streams (stdout and stderr). This approach is broadly applicable, as almost all programming languages support standard input and output.

Testing itself treats submissions as black boxes by streaming input data via standard input, capturing output data via standard output and standard error, and running standalone test oracles to validate the correctness of the output data generated by the submission (figure 3.1). Teachers can describe the task specification of input/output exercises to students, independent of any programming language. Such a task description only specifies the formatting of input and output data, and prescribes how input data must be transformed into output data.

The black-box and weakly typed nature of input/output testing comes with serious pedagogical downsides. Standard input and output are the only interfaces for reaching into and revealing internal behaviour of the submission. While other data flows and user interactions are possible in software applications (Khorram 2022), they cannot be used here. As students can only use the execution entry point (often the main function), only the behaviour of the submission as a whole can be tested. Individual steps (e.g. functions or classes) are not reachable. Feedback can thus only be provided at a holistic level.

Binary data via standard input and output are not fun, as anyone who has tried it will attest to.

Although standard input and output may in theory consume and produce binary data, programming exercises commonly use text-formatted input and output data. Implementing the interface thus involves parsing a

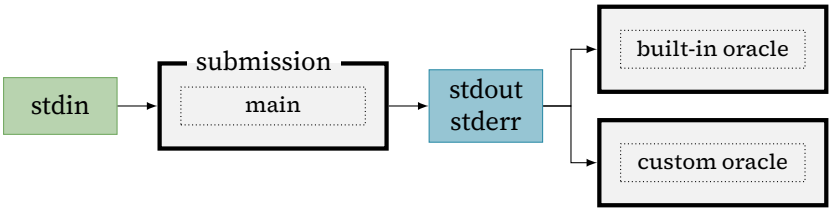


Figure 3.1. Architecture of a test execution engine based on standard input and output. The entry point of the program execution (often the main function) and standard input and output are the only interface for reaching into and revealing internal behaviour of the submission. Assessing the submission for a single test case comes down to running the executable (thick bordered boxes) with input data streamed to standard input (stdin), capturing output data from its standard output and error streams (stdout and stderr), and running standalone oracles to validate whether the output data generated by the submission satisfies the requirements from the task description.

weakly typed external string representation of input data into a strongly typed internal representation, and converting a strongly typed internal representation of output data into a properly formatted external string representation that is used for validation. As a result, test oracles have to process weakly typed output data as well.

3.1.4. Unit testing

Unit testing largely resolves these issues. Unit testing frameworks have become the norm in software development practice (Runeson 2006), and have also found their way into educational practice (Bettini et al. 2004; Ellsworth et al. 2004). Such frameworks run test cases as scripts that access internal interfaces from specific sections (units) of the submission and rely on oracles to validate that the units behave as intended. A unit could still be the entire submission (e.g. by calling the main function), but more commonly it is an individual function (in procedural or functional programming) or an individual class that is accessed through its public properties and methods (in object-oriented programming). By first writing tests for the smallest testable units, and then compound behaviours between those, one can gradually build up comprehensive tests for more complex programming exercises (Pan 1999). In addition, oracles can process strongly typed values returned by calling functions or methods or by accessing properties, allowing for more versatile correctness testing (e.g. testing real-valued numbers are correct up to an expected accuracy).

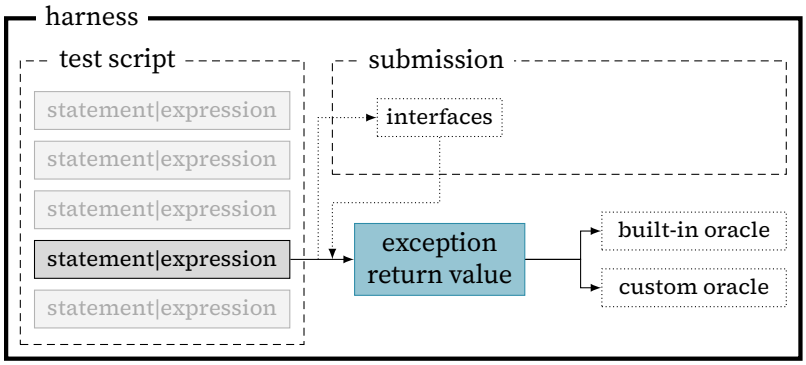


Figure 3.2. Architecture of a unit test execution engine. Assessing a submission for a single test case comes down to compiling the submission along with the test code (including test script and oracles) into an executable test harness (thick-bordered box). The test script consists of a set of statements and expressions. The harness executes the test script, whose statements and expressions may access public interfaces of a narrow section of the submitted code (unit), captures outputs (return values, exceptions) when executing statements and expressions in the script, and runs oracles to validate these outputs.

Traditional unit testing frameworks compile the submission along with the test script and oracles into a single executable called a test harness (figure 3.2). As a consequence, they often require the test script and oracles to be written in the same programming language as the submission. This restriction is rarely problematic in software development practice *sensu latu*, but hampers accepting submissions in multiple programming languages in an educational context, unless separate test suites are provided for each target language. This not only duplicates work, but unit testing frameworks for different languages also dictate how test suites are written (Agrawal and Reed 2022; Nayak et al. 2022), which differs from framework to framework.

3.1.5. TESTed 1.0

We introduced TESTed (Strijbol, Van Petegem et al. 2023; chapter 2) as an open-source educational software testing framework that accepts submissions in multiple programming languages and performs both input/output testing and unit testing, based on a single language-agnostic test suite. Currently, TESTed supports evaluating submissions in Bash, C, C#, Haskell, Java, JavaScript, Kotlin, and Python. Its abstract programming language engine transforms between language-agnostic and

language-specific representations of strongly typed values, expressions, and statements. This allows TESTed to generate language-specific test harnesses by converting language-agnostic test scripts on the fly into the programming language of a submission (figure 3.3). Executing a language-specific test harness yields strongly typed objects in a language-specific representation. The language-specific test harness then converts these objects back into abstract form before exporting them to a language-agnostic test harness for validation by oracles that run independent of the language-specific test harness.

3.1.6. Organization of this chapter

This chapter introduces TESTed-DSL as a domain-specific language (DSL) to simplify authoring test suites for language-agnostic programming exercises with support for automated assessment (section 3.2). We investigate its versatility, expressiveness, and general features by extending TESTed with support for TESTed-DSL. This yields an open-source implementation of the domain-specific language itself. It also replaces the verbose and ad-hoc JSON specification of test suites from TESTed version 1.0, which closely followed the internal representation of test suites. Statements and expressions of the abstract programming language, for example, were expressed in JSON as hierarchical structures that resemble abstract syntax trees.

We then apply TESTed-DSL to task descriptions (section 3.2.4), which allows creating task descriptions with language-specific programming interfaces (data types, literal values, identifiers, and code fragments) expressed in a language-agnostic way. A template engine then transforms the language-agnostic interface bindings into specific representations for any target programming language supported by TESTed, while preserving other formatting of task descriptions.

This is followed by a set of examples illustrating the use of TESTed-DSL (section 3.3). We then elaborate on a case study of using TESTed-DSL in educational practice (section 3.4.1) and analyse the performance of using the domain-specific language (section 3.4.2). The results and impact of our contributions are discussed at length (section 3.5). We conclude that extending TESTed with a domain-specific language for specifying test suites and task descriptions provides an expressive and ergonomic solution. It allows for authoring a diverse set of language-agnostic programming exercises that support automated assessment in educational practice. We close by drawing the roadmap for some ongoing and future work (section 3.6).

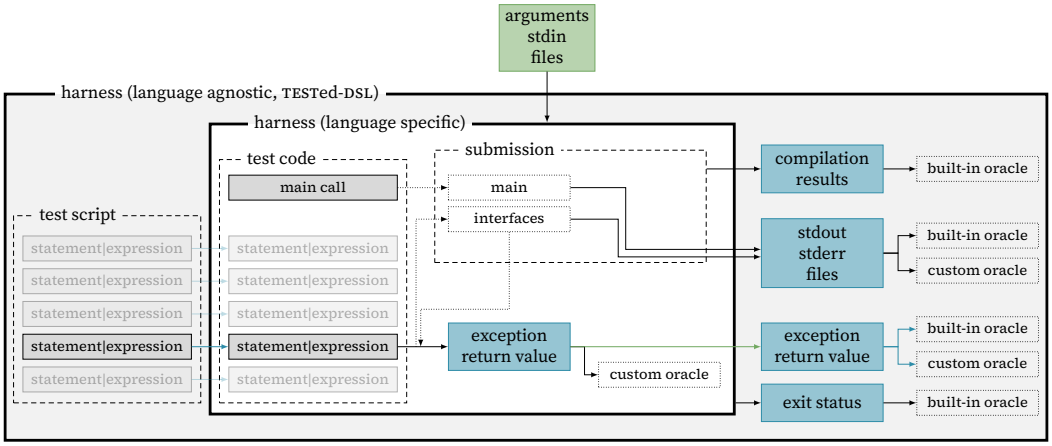


Figure 3.3. Architecture of the test execution engine of TESTed. Since TESTed supports both input/output testing and unit testing, its architecture is a combination of figures 3.1 and 3.2. TESTed achieves this by running a language-specific harness (white box) within the language-agnostic test harness (grey box). Assessing a submission for a single test case again comes down to compiling the submission along with test code into an executable test harness. The input data are passed to the executable, and the test script consists of a set of statements and expressions. However, the strongly typed values, expressions and statements in the test script may now be in a language-agnostic representation. Before compilation, the test script is transformed (blue arrows) into language-specific test code. While the main call interacts with the main function of the submission, other tests access public interfaces of the submission (a single statement/expression is illustrated here). If a language-specific oracle is used, it is included in the language-specific harness. Otherwise, the strongly-typed outputs, together with the outputs from the main function, are processed in the language-agnostic harness. To make this possible, strongly-typed outputs are transformed from a language-specific representation into a language-agnostic representation (green arrow). Finally, oracles in the language-agnostic harness evaluate the outputs. To make this possible, strongly typed values are converted in to the language-specific representation of the oracle (blue arrow).

```yaml
- unit: "Greet function"
  cases:
    - expression: "greet('World')"
      return: "Hello, World!"
```

Listing 3.1. *Hello, World!* example of a test suite in TESTed-DSL. The test suite has a single test case that calls the function `greet` with string argument `"World"`. The function is expected to return the string `"Hello, World!"`.

3.2. TESTed-DSL

TESTed-DSL was conceived as a domain-specific language for language-agnostic test suites of programming exercises that support automated assessment. Such a test suite generally consists of two parts: individual tests (including test data) and a structured grouping of tests into units. In this section, we first introduce the structure of the domain-specific language, followed by the abstract programming language that is used to represent test data, expressions, and statements.

3.2.1. Test suite structure

TESTed-DSL uses YAML (Ben-Kiki et al. 2021) as a markup language for describing test suites that are preferably specified in a language-agnostic way. The structure of test suites is formally specified in a JSON Schema.[1] Note that this schema can be plugged into text editors or integrated development environments for syntax highlighting, autocompletion, and validation. Listing 3.1 shows a minimal example of a test suite. More elaborate examples are discussed in section 3.3.

The test suites have a hierarchical structure: a test suite may have multiple **units**. Each unit is tested by multiple **test cases**. A test case consists of **setup** code, an optional **main call**, a **script**, and **teardown** code. The minimal example has a single unit with a single test case, and no setup, main call, nor teardown. The main call and the script together comprise the **tests** for the test case (figure 3.3). The structure of test suites using the domain-specific language is identical to JSON test suites (section 2.6.1), but the terminology used by the domain-specific language aligns more with the literature.

The Dodona terminology is also supported (using tab, context, test case, and test respectively).

The input data for the main call is made available as **files**, passed as **arguments**, streamed through standard input (**stdin**), or any combination

[1] `https://github.com/dodona-edu/universal-judge/blob/master/tested/dsl/schema.json`

thereof. The input data for the script are the **statements** and **expressions** it consists of. The minimal example has a single script with a single test expression. However, in TESTed-DSL, there is no separation between these types of tests: all tests (the main call and the script) are given as individual tests of a test case. The type of the test is automatically derived from the available input. If standard input or arguments are present, the test is considered to be a main call and will be executed as such.

When TESTed runs a test, it catches any **runtime exception** and output sent through the standard output streams (**stdout** and **stderr**). It also catches the **return value** when expressions are evaluated (as is done in the minimal example) and the **exit status** when the process running the language-specific test harness terminates.

Test suites can specify a value for each possible input and an expected value for each possible output of a test. Most inputs and outputs of tests are weakly typed: arguments, standard input and output, (text) files, and messages of runtime exceptions are always strings, and the exit status is always an integer. Return values, on the other hand, are strongly typed.

The validation of the actual values against the expected values happens with oracles. By default, the built-in oracles of TESTed are used with default expected values. For example, an exit code must be 0, and there must not be any output on standard error. To avoid unnecessary feedback, correct tests against a default value are not reported. For example, if the exit code of the submission is not zero, this will be reported as a failure. However, it is distracting to show a successful zero exit code for all test cases (e.g. even those that test function calls). For that reason, the exit code test is typically hidden. In the minimal example, the exit code would be checked, but not reported unless it was non-zero.

For all outputs, the built-in oracles perform equality testing: the actual value must match the expected value. However, the built-in oracles feature adjustable parameters that offer a degree of flexibility in this equality testing. Examples include case-insensitive comparisons for strings and a tolerance for slight inaccuracies in real number comparisons. Importantly, these parameter settings adhere to an inheritance mechanism across the test suite hierarchy. Setting a parameter at a certain node within the suite takes precedence over any similar setting at a higher level and extends to all subordinate nodes.

Custom oracles can also overrule built-in oracles for standard output streams (stdout and stderr) and return values. Custom oracles are called with the expected and actual values, in addition to some metadata about the test. These allow for custom validations beyond equality testing.

They can also be useful for non-deterministic results, e.g. results that depend on the current date.

3.2.2. Abstract programming language

TESTed-DSL adopts a subset of the Python programming language as the language-agnostic representation of expressions, statements, and strongly typed values. The subset is deliberately chosen to suit the specific needs of language-agnostic test suites, and therefore does not encompass all Python features.

An expression in the abstract programming language may consist of literals, identifiers, function calls, constructors, method calls, and property access. Functions, constructors, and methods take expressions as positional arguments or as named arguments. A statement is either an assignment or an expression. Unlike other programming languages, Python has no way to differentiate constructors from function calls. Because that difference is important to make when transforming language-agnostic expressions into the specific syntax of some programming languages (e.g. using the `new` keyword), TESTed-DSL follows the convention that constructors call a function whose name begins with a capital. Another convention is that variables that are all caps are considered global variables. Otherwise, TESTed-DSL follows Python conventions where possible.

A strongly typed value is obtained when TESTed evaluates a test expression. These return values can be denoted as native YAML scalars (`null`, `booleans`, `integers`, `real numbers`, and `strings`), `sequences`, `sets` (using the `!!set` tag), and `mappings`. By default, TESTed-DSL resolves these YAML objects as basic types of TESTed. Casting to advanced types of TESTed is possible by using explicit YAML data types (`!type` tag). As such, an expected return value `42` with TESTed data type `int64` can be expressed as `!int64 42`.

YAML strings are interpreted as literal strings by default, but are interpreted as expressions in the abstract programming language when the `!expression` tag is used. As a result, **`!set`** `[1, 2, 3]` can also be denoted as **`!expression`** *`"set([1, 2, 3])"`*.

3.2.3. Language-specific test suites

As the abstract representation of TESTed-DSL is independent of any programming language, not every feature and data type of every program-

ming language is supported. Examples include object equality checking (in object-oriented languages) or pointers (in C). To accommodate this, TESTed-DSL also supports language-specific representations. The language-specific representations are copied verbatim into the language-specific test harness and bypass the language-agnostic harness. This means that all language features of the programming languages can be used.

In addition to language-specific statements and expressions, TESTed also allows writing custom oracles for return values whose expected data type is not supported by the abstract representation of TESTed. Because such oracles run inside the language-specific harness, they also bypass the need for transforming return values, as is the case for traditional language-specific unit testing frameworks.

However, these language-specific features have to be used sparingly, as they restrict testing to programming languages for which language-specific representations and custom oracles are specified. Listing 3.6 contains an example of a test suite that uses language-specific representations.

3.2.4. Language-agnostic task descriptions

For automated assessment to work, task descriptions of programming exercises have to specify what the submissions must implement (the interface) and how submissions must behave (interactions with the interface). The expected behaviour prescribes what type of input data is passed when interfaces are accessed, what type of output data they must return and how they need to transform input data into output data. When references to and interactions with these interfaces have specific bindings to programming languages, they result in language-specific task descriptions. For example, if a submission must implement a function that filters a list, the naming convention of the function and the values passed to this function (a list) will look different depending on the programming language.

The same rationale for having a single, language-agnostic test suite also applies to task descriptions. A similar approach to avoid the need to author language-specific task descriptions for each target programming language of an exercise can be followed. There are some differences though. First of all, language-specific interface references and interactions in task descriptions are usually embedded in natural language content formatted with some plaintext markup language (e.g. Markdown, HTML or reStructuredText). We therefore extended TESTed with an engine

that supports authoring task descriptions as Jinja2 templates (Ronacher and Lord 2022) with language-agnostic interface bindings embedded as `{{binding}}` placeholders in plain-text documents. The engine then automatically transforms each placeholder into a specific representation for any target programming language supported by TESTed, while preserving surrounding formatting of task descriptions.

For each type of interface with language-specific bindings, the engine provides a corresponding Python variable or function that identifies the interface. Interface functions take a language-agnostic representation (a string) of the interaction as their first argument and return its corresponding language-specific representation (also a string) that replaces the placeholder.

Interface interactions in task description templates are natural counterparts of their representations in TESTed-DSL test suites: **statements**, **expressions**, and **literal values**. However, in task description templates, the engine makes no difference between statements, expressions, and literal values. All are represented as a statement. To reference a named function, the snake case version of its identifier is passed to the function identifying the interface in TESTed-DSL's abstract programming language: variables (`variable`), functions (`function`), classes (`class`), methods (`method`), properties (`property`) and parameters (`parameter`). The engine formats these according to the naming conventions for the target programming language. For example, `function('the_function')` will result in the string `TheFunction` in C#.

In addition, the engine also exposes a `datatype` function, which can be passed the name of a basic or advanced data type. For example, `datatype('integer')` denotes the basic type `integer`. This function returns an object, that when converted to a string by the engine, results in the formal name in the target programming language. The object also supports two properties: `singular` and `plural`. These methods return the informal name for the type (in singular or plural form respectively) in the target programming language.

The engine also exposes some additional environment variables: the target programming language (`language`), the target natural language (`natural_language`) and the namespace (`namespace`) that is specified for some languages. These environment variables are particularly useful in combination with Jinja2 control structures, for example, to add language-specific sections to task descriptions.

Task description templates can also reuse the language-agnostic specifications of TESTed-DSL test suites for code fragments that illustrate the

expected behaviour of interfaces with example interactions corresponding to test cases. The template engine renders these fragments in the style of Python doctests: language-specific statements with language-specific string representations of expected outputs. In task description mode, the engine ignores features of test suite specifications that steer the testing process. For example, task descriptions only need expected outputs of interface interactions and no oracles to effectively validate their correct behaviour. TESTed has special support for including these test suites in Markdown files as code blocks: ```` ```dsl ... ``` ````.

3.3. Illustrative examples

This section contains some examples of language-agnostic test suites and task descriptions.

3.3.1. Language-agnostic test suites

Rather than showcasing all features supported by TESTed-DSL, the examples want to give an idea of what the YAML test suites look like. We begin with the most common scenario: a language-agnostic test suite that relies on built-in oracles. Next, we provide two more advanced test suites that illustrate how advanced data types and custom oracles are handled in the DSL, while also illustrating the versatility of TESTed-DSL in describing test suites for different types of programming exercises.

Listing 3.2 illustrates the hierarchical structure of units, test cases, and tests (in a script). The test suite has a single unit `Cipher` (line 2) containing all test cases to validate the correct behaviour of submissions that must define the class `Cipher`.[2]

The unit has multiple test cases, but only a single test case (lines 4–28) is shown completely for illustrative purposes. The first test case does not provide inputs that require scheduling a main call, as it only expects submissions to implement a class definition. The first statement of its script calls the constructor of the class with two string arguments, instantiating an object that is assigned to the variable `cipher01` (line 5). The next two expressions access the properties `grid` (line 6) and `map` (line 12) of the object to validate they are properly initialized by the constructor. The expected value of the `grid` property is specified as a nested sequence of strings that represents a 4×4 grid of characters (lines 7–11). Note

[2] `https://dodona.be/en/activities/636251211/`

that we have used YAML block style notation (using dashes) for the outer sequence and YAML flow style notation (elements enclosed in square brackets and separated by commas) for the inner sequences of the expected value. The expected value of the `map` property is specified using the YAML block style notation of a mapping from strings onto strings (lines 13–20).

The next two expressions call the methods `encode` (line 21) and `decode` (line 23) on the object with a single string argument, and are expected to return a string value (lines 22 and 24). The last two expressions call the same methods `encode` (line 25) and `decode` (line 27) with other string arguments, and are now expected to raise an exception with the message `invalid message` (lines 26 and 28). This test suite provides language-agnostic representations for all statements, expressions, and strongly typed values, so TESTed can use it to validate submissions in any supported programming language (including future supported languages).

TESTed-DSL makes an important distinction between expressions and statements (`expression` and `statement`). The evaluation of expressions returns a strongly typed value that is captured by the language-specific test harness. The default expected value is any object having data type `nothing` (such as `null` or `None`). However, the `return` attribute can be used to specify the expected value in the test suite. Statements, on the other hand, are executed by the language-specific test harness. Even though the TESTed-DSL statement might actually be an expression in some target languages (like the assignment in line 5), TESTed will accept any return value resulting from executing the statement (other outputs are still captured and tested). This behaviour cannot be altered: TESTed-DSL does not allow specifying an expected return value and/or a custom return oracle for statements.

TESTed is designed in part to run as a standalone command line tool to generate structured feedback on standard output. It is also well-equipped for seamless integration with online learning platforms that display task description of programming exercises, accept student submissions, run TESTed in their backend and render the resulting feedback. TAP (Schlueter et al. 2022) (Test Anything Protocol) is the *de facto* standard used by unit testing frameworks for reporting test results. For educational purposes, however, TESTed streams richer and more structured feedback to standard output, in a format formally specified in a JSON Schema.[3] Dodona (Van Petegem, Maertens et al. 2023), for example, renders each unit of test cases (from listing 3.2) in a separate tab and visually groups all tests from the same test case inside a card (figure 3.4).

[3] `https://github.com/dodona-edu/dodona/blob/main/public/schemas/partial_output.json`

```
1  units:
2   - unit: 'Cipher'
3     cases:
4       - script:
5           - statement: "cipher01 = Cipher('ABCD', '1AX3S1M2PYZ')"
6           - expression: "cipher01.grid"
7             return:
8               - ['-', 'A', 'X', '-']
9               - ['-', '-', 'S', '-']
10              - ['M', '-', '-', 'P']
11              - ['Y', 'Z', '-', '-']
12          - expression: "cipher01.map"
13            return:
14              'A': 'AB'
15              'X': 'AC'
16              'S': 'BC'
17              'M': 'CA'
18              'P': 'CD'
19              'Y': 'DA'
20              'Z': 'DB'
21          - expression: "cipher01.encode('spam')"
22            return: 'BCCDABCA'
23          - expression: "cipher01.decode('BCCDABCA')"
24            return: 'SPAM'
25          - expression: "cipher01.encode('eggs')"
26            exception: 'invalid message'
27          - expression: "cipher01.decode('BCCDBACA')"
28            exception: 'invalid message'
29      - script:
30          # statements and expressions of the second script
```

Listing 3.2. Language-agnostic test suite to validate the correct behaviour of submissions that must define the class `Cipher`, whose objects have properties `grid` and `map`, and methods `encode` and `decode`. Only the first test case is shown completely for illustrative purposes. Because this test suite only has a single unit, the `units` (line 1) could be removed, making the list of units the top-level construct in the test suite.

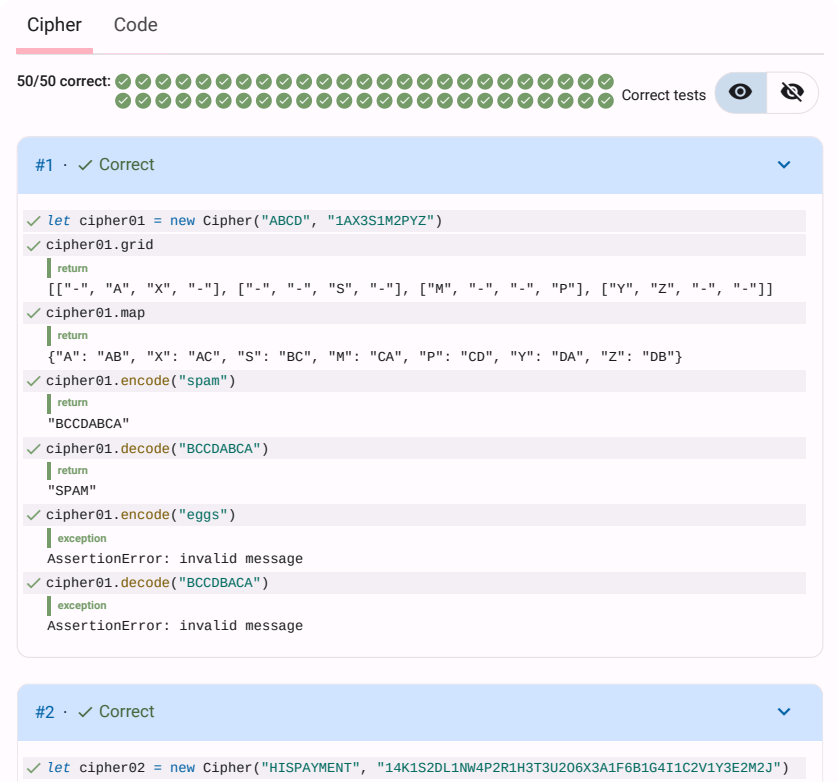


Figure 3.4. Dodona rendering of test results after TESTed validated a correct JavaScript submission for the programming exercise configured with the language-agnostic test suite from listing 3.2. All tests from a single test case are visually grouped inside a card. Green check marks show that all tests of each script succeeded. Each expression has one reported output – either its return value (`return`) or a runtime exception (`exception`) – that is validated to be correct as indicated by the green colour of the header for the reported output.

Because JavaScript code was submitted in this case, TESTed adopted the syntax and conventions configured in its JavaScript module to format all statements, expressions, and return values in the feedback. The JavaScript-specific syntax highlighting is done by Dodona as part of the process of rendering the feedback.

Parallel to dynamic testing, TESTed flags programming errors, bugs, stylistic errors and suspicious constructs by running a language-specific linter (see section 2.7.6) on each submission (Truong et al. 2005). Dodona displays these linter messages inline in the source code of the submission, in a separate tab called Code. Linters are preconfigured in the language-specific modules of TESTed, so no additional configuration is needed in the test suite. Test suites may, however, overrule these linter configurations.

Recoupling exercise: multiple functions and advanced data types

Listing 3.3 shows a language-agnostic test suite for an exercise that asks to implement two functions: `divide` and `recouple`.[4] The first function must divide the given string into a number of parts. The second function must split each given string and then recombine the corresponding parts into new strings. As part of the problem-solving process, students may discover a divide-and-conquer strategy: the implementation of the second function may call the first function that solves a subtask.

However, the test suite validates the correct behaviour of both functions in two separate units. This example illustrates that TESTed-DSL allows leaving out the grouping of tests in a script as a shorthand for the common case of test cases whose script has a single test. The first two test cases for the `divide` function (lines 4–7) use the `return` attribute with flow style notation of a YAML sequence to specify the function call is expected to return a sequence of strings as basic types of TESTed. As a result, both lists and tuples will, for example, be accepted for Python submissions. On the contrary, the first two test cases for the `recouple` function (lines 11–14) use an explicit cast to force the return value to be a list (line 12) or a tuple (line 14) for languages like Python that make the difference. For Java and JavaScript submissions, however, arrays are accepted in both cases. The same observation holds for the first argument passed to the function: a list (line 11) or a tuple (line 13) for languages that make the difference, or the default sequence type for other languages.

Separating validation of the two functions across two separate units allows students to immediately pinpoint what functions already behave

[4] `https://dodona.be/en/activities/1145516160/`

```
units:
  - unit: "Divide"
    scripts:
      - expression: "divide('accost', 3)"
        return: ["ac", "co", "st"]
      - expression: "divide('COMMUNED', 4)"
        return: ["CO", "MM", "UN", "ED"]
      - expression: "divide('programming', 5)"
        exception: "invalid division"
  - unit: "Recouple"
    scripts:
       - expression: "recouple(['ACcoST', 'COmmIT', 'LAunCH',
       ↪  'DEedED'], 3)"
         return: !list ["ACCOLADE", "communed", "STITCHED"]
       - expression: "recouple(('ACCOLADE', 'communed',
       ↪  'STITCHED'), 4)"
         return: !tuple ["ACcoST", "COmmIT", "LAunCH", "DEedED"]
       - expression: "recouple(['programming', 'computer',
       ↪  'games'], 5)"
         exception: "invalid division"
```

Listing 3.3. Language-agnostic test suite to validate correct behaviour of submissions that must define the functions `divide` and `recouple`. Because each test case has a single test, grouping of tests in a script can be left out from the test suite specification as a shorthand.

as expected from the feedback. Figure 3.5 shows a Dodona rendering of the generated feedback for a Python submission whose implementation of the first function passes all tests. However, the second function has three tests that fail for different reasons. The first function call returns the correct result, but also writes the string `spam` to standard output (`stdout`) whereas no output is expected on this output stream. The second function call returns a list where a tuple was expected. The third function call should throw an exception, but this does not happen.

Sum of three cubes exercise: input/output testing and custom oracles

Listing 3.4 shows a test suite for an input/output exercise that asks to read an integer k from standard input and write a solution of the sum of three cubes problem ($x^3 + y^3 + z^3 = k$) to standard output as three lines containing non-zero integers x, y and z (A. R. Booker 2019). The test suite has one unit with three test cases whose script only specifies a single line of input streamed into standard input of the main call and three expected lines of output printed on standard output. So again the shorthand structure applies here for units and scripts.

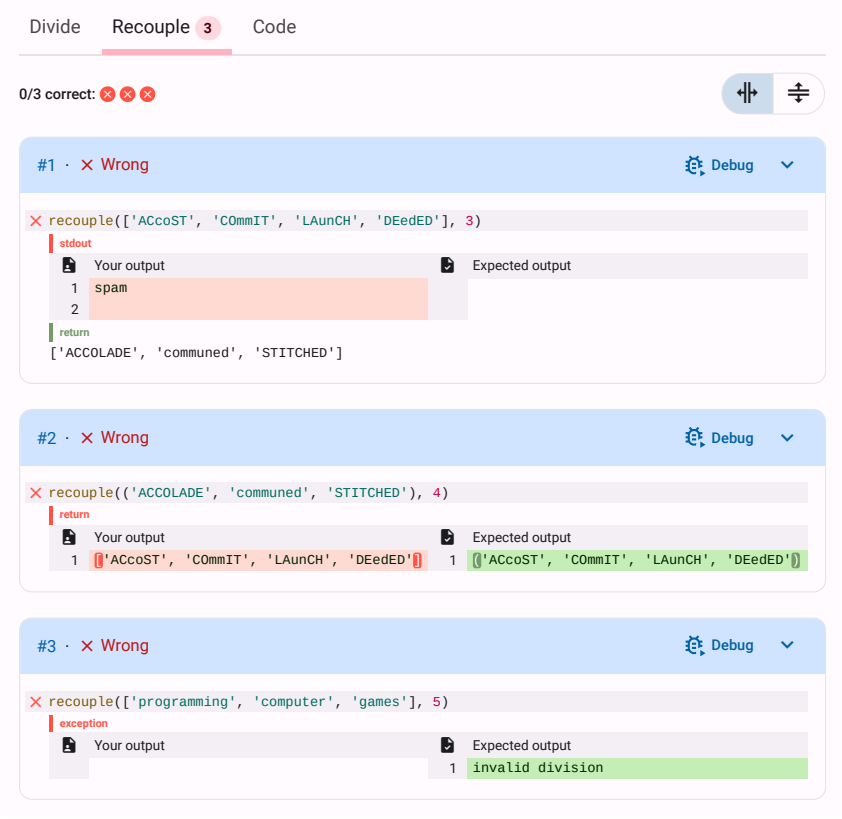


Figure 3.5. Dodona rendering of test results after TESTed validated a wrong Python submission for a programming exercise configured with the language-agnostic test suite from listing 3.3. All test cases succeed for the implementation of the function `divide`, but for different reasons, some test cases fail for the implementation of the function `recouple`. An extra badge in the tab header displays the number of failing test cases in the corresponding unit, or the number of source code annotations in case of the Code tab (none in this case).

```
1  - unit: "Sum of three cubes"
2    scripts:
3      - stdin: "3"
4        stdout: "1\n1\n1\n"
5      - stdin: "33"
6        stdout: |
7          8866128975287528
8          -8778405442862239
9          -2736111468807040
10     - stdin: "42"
11       stdout:
12         data: |
13           -80538738812075974
14            80435758145817515
15            12602123297335631
16         oracle: custom_check
17         name: "sum_of_three_cubes"
18         file: "oracle.py"
19         arguments: [42]
```

Listing 3.4. Language-agnostic test suite to validate correct behaviour of submissions for an input/output exercise that asks to read an integer k from standard input and write a solution of the sum of three cubes problem $x^3 + y^3 + z^3 = k$ to standard output as three lines containing non-zero integers x, y and z.

The first two test cases show different YAML alternatives for specifying multi-line strings (line 4 and lines 6–9) as the expected value streamed to standard output. However, specifying a fixed expected output is problematic for this exercise. The task description does not imply any order in which the three integers must be listed, so any permutation of the three integers given by the expected solution for the second test case should also be validated as a correct solution. Moreover, some values of k have alternative solutions, irrespective of permutations. For example, another way to solve the first test case is given by $(-5)^3 + 4^3 + 4^3 = 3$ (Sutherland and A. Booker 2019).

As enumerating all possible solutions (and their permutations) is infeasible, the specification of the third test case provides a better approach. TESTed uses the function `sum_of_three_cubes` from the Python module `oracle.py` as a custom oracle to validate the correctness of the output generated on stdout. TESTed always passes the actual output and some metadata, such as the programming language, and the expected output as the first argument to the oracle. Additionally, extra arguments taken from the test suite can be passed (in this case there is only one extra argument: the integer 42).

The custom oracle needs to *i*) check the output string has the correct

format (three lines containing one integer each), *ii*) parse the three integers x, y and z from the output (converting their string representation into integers) and *iii*) check the integer expression $x^3 + y^3 + z^3$ yields the value of the second argument (`42`). Passing an extra argument is not strictly needed here as the custom oracle could also derive the data from the expected value, using the same procedure as used to derive the data from the actual value.

3.3.2. Language-agnostic task descriptions

Listing 3.5 shows a language-agnostic task description for the *Recoupling* exercise that was introduced in section 3.3.1. The task description is specified using Kramdown-flavored Markdown. It contains Jinja2 placeholders (`{{code}}`) for function names and formal/informal names of data types, along with MathJax placeholders (`$$...$$`) for LaTeX formulae. The task description ends with some examples of the expected behaviour when calling the two functions `divide` and `recouple` that must be implemented for this programming exercise. The language-agnostic specification of the sample code is denoted using the same specification of the test suite for the programming exercise in TESTed-DSL format or a reduced version thereof (see listing 3.3).

For the sample code in the template, we use the Jinja2 import facilities to include the test suite from listing 3.3. TESTed will automatically generate an example based on this test suite, while ignoring structural elements (like the hierarchy).

Starting from a task description template, the TESTed template engine can generate language-specific versions of the task description for all supported programming languages (figure 3.6). This is done by taking into account language-specific conventions (naming, quoting, formal and informal names and syntax for literals, expressions, and statements) as specified in the language modules for the supported programming languages. This dual use between test suites and task description keeps the language-specific modules of TESTed lightweight and guarantees consistency between the generation of language-specific tests and descriptions.

3.4. Evaluation

During the academic year 2023–2024 we started promoting TESTed with TESTed-DSL as the primary way to author new (language-agnostic) pro-

```
Write a function `{{function('divide')}}` that takes two arguments:
↪  _i_) a word (`{{datatype('text')}}`) and _ii_) the number of
↪  (non-overlapping) groups $$n \in \mathbb{N}_0$$
↪  (`{{datatype('integer')}}`) into which the word must be
↪  divided. If the word passed to the function
↪  `{{function('divide')}}` cannot be divided into $$n$$ groups
↪  that have the same length, an exception must be raised with the
↪  message `invalid division`. Otherwise, the function must return
↪  a {{datatype('list').singular}} (`{{datatype('list')}}`)
↪  containing the $$n$$ groups (`{{datatype('text')}}`) into which
↪  the given word can be divided. All groups need to have the same
↪  length (same number of letters).

Write another function `{{function('recouple')}}` that takes two
↪  arguments: _i_) a {{datatype('sequence').singular}}
↪  (`{{datatype('sequence')}}`) of $$m \in \mathbb{N}_0$$ words
↪  (`{{datatype('text')}}`) and _ii_) the number of
↪  (non-overlapping) groups $$n \in \mathbb{N}_0$$
↪  (`{{datatype('integer')}}`) into which the words must be
↪  divided. If at least one of the words passed to the function
↪  `{{function('recouple')}}` cannot be divided into $$n$$ groups
↪  that have the same length, an exception must be raised with the
↪  message `invalid division`. Otherwise, the function must return
↪  a {{datatype('sequence').singular}} containing the $$n$$ new
↪  words (`{{datatype('text')}}`) obtained when each of the $$m$$
↪  given words is divided into $$n$$ groups that have the same
↪  length, and if each of the $$m$$ corresponding groups is merged
↪  into a new word. The type of the returned
↪ {{datatype('sequence').singular}} (`{{datatype('sequence')}}`)
↪  must correspond to the type of the
↪  {{datatype('sequence').singular}} passed as a first argument to
↪  the function.

### Example

```dsl
{% include 'tests.yaml' %}
```
```

Listing 3.5. Language-agnostic task description for the *Recoupling* exercise. The Kramdown-flavored Markdown contains Jinja2 placeholders for function names and both formal and informal names of data types. The task description ends with some examples that illustrate how the two functions should be used. The sample code is denoted using the same specification of the test suite for the programming exercise in TESTed-DSL format (listing 3.3).

Recoupling

Write a function `divide` that takes two arguments: *i*) a word (`str`) and *ii*) the number of (non-overlapping) groups $n \in \mathbb{N}_0$ (`int`) into which the word must be divided. If the word passed to the function `divide` cannot be divided into n groups that have the same length, an exception must be raised with the message `invalid division`. Otherwise, the function must return a list (`list`) containing the n groups (`str`) into which the given word can be divided. All groups need to have the same length (same number of letters).

Write another function `recouple` that takes two arguments: *i*) a sequence (`list` or `tuple`) of $m \in \mathbb{N}_0$ words (`str`) and *ii*) the number of (non-overlapping) groups $n \in \mathbb{N}_0$ (`int`) into which the words must be divided. If at least one of the words passed to the function `recouple` cannot be divided into n groups that have the same length, an exception must be raised with the message `invalid division`. Otherwise, the function must return a sequence containing the n new words (`str`) obtained when each of the m given words is divided into n groups that have the same length, a

is merged into a new word. The type of the returne

correspond to the type of the sequence passed as

Example

```
>>> divide('accost', 3)
['ac', 'co', 'st']
>>> divide('COMMUNED', 4)
['CO', 'MM', 'UN', 'ED']
>>> divide('programming', 5)
Exception: invalid division

>>> recouple(['ACcoST', 'COmmIT', 'LAunCH', 'DEedE
['ACCOLADE', 'communed', 'STITCHED']
>>> recouple(('ACCOLADE', 'communed', 'STITCHED'),
('ACcoST', 'COmmIT', 'LAunCH', 'DEedED')
>>> recouple(['programming', 'computer', 'games'],
Exception: invalid division
```

Recoupling

Write a function `divide` that takes two arguments: *i*) a word (`string`) and *ii*) the number of (non-overlapping) groups $n \in \mathbb{N}_0$ (`number`) into which the word must be divided. If the word passed to the function `divide` cannot be divided into n groups that have the same length, an exception must be raised with the message `invalid division`. Otherwise, the function must return a list (`array`) containing the n groups (`string`) into which the given word can be divided. All groups need to have the same length (same number of letters).

Write another function `recouple` that takes two arguments: *i*) a sequence (`array`) of $m \in \mathbb{N}_0$ words (`string`) and *ii*) the number of (non-overlapping) groups $n \in \mathbb{N}_0$ (`number`) into which the words must be divided. If at least one of the words passed to the function `recouple` cannot be divided into n groups that have the same length, an exception must be raised with the message `invalid division`. Otherwise, the function must return a sequence containing the n new words (`string`) obtained when each of the m given words is divided into n groups that have the same length, and if each of the m corresponding groups is merged into a new word. The type of the returned sequence (`array`) must correspond to the type of the sequence passed as a first argument to the function.

Example

```
> divide("accost", 3)
["ac", "co", "st"]
> divide("COMMUNED", 4)
["CO", "MM", "UN", "ED"]
> divide("programming", 5)
Error: invalid division

> recouple(["ACcoST", "COmmIT", "LAunCH", "DEedED"], 3)
["ACCOLADE", "communed", "STITCHED"]
> recouple(["ACCOLADE", "communed", "STITCHED"], 4)
["ACcoST", "COmmIT", "LAunCH", "DEedED"]
> recouple(["programming", "computer", "games"], 5)
Error: invalid division
```

Figure 3.6. Dodona rendering the Python (top left) and JavaScript (bottom right) versions of the task description as generated by the template engine of TESTed.

Table 3.1. Dodona programming exercises using TESTed for automated assessment.

Programming language	№ of exercises	№ of submissions
Bash	32	121 011
C	83	652
C#	1	6
Haskell	86	241
Java	120	1493
JavaScript	271	178 011
Kotlin	146	271
Python	385	31 510
Total	**1124**	**333 195**

gramming exercises on Dodona (Van Petegem, Maertens et al. 2023). We clearly documented the process, including guides and reference documentation.[5] At the time of writing, 1124 programming exercises on Dodona support automated assessment through TESTed, with 333 195 assessed student submissions (table 3.1).

In the next sections, we look into a case study where we converted all JavaScript exercises of a course to use TESTed-DSL. This resulted in 72 programming exercises and 22 018 student submissions (these are part of the total numbers shown in table 3.1).

3.4.1. Expressiveness and ergonomics

To evaluate the expressiveness of TESTed-DSL and its applicability in educational practice, we authored test suites for a collection of 72 programming exercises that were used during the 2023 spring semester in an introductory course at Ghent University taken by 114 students.[6] None of these exercises is based on input/output, 22 ask to implement one or more functions and 50 ask to implement one or more classes.

The largest part of the collection consists of all the 66 JavaScript exercises we designed for this course over the years. These exercises already supported automated assessment with a test suite designed for a custom JavaScript-specific testing framework. To migrate automated assessment for these exercises to TESTed, we wrote a script that either converts their existing JavaScript test suites into a TESTed-DSL specification or reports why this is not possible. Seamless automated conversion was

[5] `https://docs.dodona.be/nl/guides/exercises/` (some documentation is currently only available in Dutch).

[6] `https://dodona.be/en/courses/2263/`

helped by the fact that we designed TESTed according to best practices we obtained from designing and implementing many language-specific testing frameworks for Dodona, and a generic feedback format (JSON) that closely follows the structure of TESTed-DSL test suites.

All existing test suites for these JavaScript exercises could be transformed into TESTed-DSL: 58 of 66 (88 %) into language-agnostic test suites and 8 of 66 (12 %) into JavaScript-specific test suites. The latter use JavaScript-specific language constructs that are not (yet) supported by TESTed: array indexing, object indexing, array destructuring assignment, object identity checking (=== operator), class constants, spread syntax (. . . operator), and the use of arrow functions as callbacks. The sample solutions for the programming exercises – assumed to be correct – were used for testing the test suite migration throughout the process.

Although the above experiment made clear that most, but not all, test suites for existing JavaScript exercises could be converted into language-agnostic test suites for TESTed, TESTed-DSL was expressive enough to also specify JavaScript-specific test suites for the remaining exercises. At least we may conclude from this experiment that for our practice, we can gradually deprecate language-specific testing frameworks in favour of TESTed. Along the way, converting test suites into a language-agnostic TESTed-DSL specification also broadens the usability of the programming exercises across programming languages.

We also added 6 new programming exercises with language-agnostic TESTed-DSL test suites to the collection, designed from scratch and used for a midterm test (2 exercises) and for 2 exam sessions (2 exercises each) during the semester. Our experience in doing so is that direct authoring of TESTed-DSL test suites is ergonomic, either when done by hand or using a generator script based on a correct solution. Human error is reduced by using the formal specification for automatic verification that TESTed-DSL test suites are well-formed and valid. TESTed performs such verification while parsing test suites, but this can also be done interactively while authoring the test suites in an IDE. For example, we provide a VS Code Plugin that automatically applies the JSON Schema to the YAML files to catch errors and to provide autocompletion.[7] The improved readability of TESTed-DSL test suites further increases productivity and enhances comfort while authoring programming exercises, especially with editor support for syntax highlighting and autocompletion.

Of these exercises, 19 were used as mandatory assignments during the 2023 spring semester of the introductory course and 6 for midterm tests

[7] https://marketplace.visualstudio.com/items?itemName=dodona.dodona-exercise-plugin

and exams. Students could use the remaining exercises from the collection for additional practice in preparation for tests and exams. Students were not restricted in the number of submissions for each exercise. They received immediate feedback from automated assessment upon each submission, also after the submission deadline and also during midterm tests and exams. Students had previous experience in working with Dodona for solving Java, Python and Bash programming exercises with automated assessment. But most of them had no prior experience with the collection of JavaScript exercises, nor with automated assessment based on the custom JavaScript-specific testing framework used in previous editions of the course.

For each course edition, we make a different selection of mandatory exercises and we always design new exercises for midterm tests and exams. This way, students who had to retake the course had no reference for comparison between the custom JavaScript-specific testing framework used in previous course editions and TESTed using during the 2023 edition. For that reason, we decided not to inform students about us switching the JavaScript testing framework that for them runs as a hidden component in the backend of Dodona.

In total, TESTed automatically assessed 22 018 JavaScript submissions during the 2023 spring semester of the course. None of the questions we received from students during the hands-on sessions for the course or via Dodona's Q&A module hinted at any issues with automated assessment or the feedback it generated other than content-related issues. We also found no indications of problems in the course evaluation that students complete some weeks after the end-of-semester exams.

3.4.2. Performance

As is the case for software testing in general, the user experience of educational software testing depends on the performance of test runs to limit how long students must wait before feedback is reported (Sarsa et al. 2022). We therefore ran a benchmark using all 72 JavaScript exercises mentioned previously to validate the performance of TESTed. These exercises have 95 test cases on average, with a minimum of 2 and a maximum of 518. Previously, we showed that the extra flexibility provided by language-agnostic testing comes with an acceptable overhead when dynamically generating test code for language-specific test harnesses (section 2.10.3). Here, we specifically investigated if using a domain-specific language for specifying test suites incurs additional overhead compared to using the JSON-formatted test suite specifications

introduced in version 1.0 of TESTed. The latter specifications closely reflect the internal structure and functionality of the testing framework, whereas TESTed-DSL was designed to make test suite authoring as ergonomic as possible.

The benchmark was run on a laptop running Linux (NixOS version 24.05) with 32 GiB RAM, an Intel i7-11850H processor, and a 1 TiB SSD. No power-saving features were active during the benchmark. It turns out that parsing and processing of TESTed-DSL test suites takes 20 % less time on average compared to the original JSON-formatted test suites. We believe this is mainly due to using a more performant YAML parsing library compared to the Python built-in JSON parsing library. However, this illustrates TESTed-DSL does not incur any additional overhead compared to the original JSON format. Because the domain-specific language still supports all features of TESTed, we decided to deprecate support for JSON-formatted test suites in favour of TESTed-DSL.

Equally important is the total runtime for assessing the submissions for a programming exercise. This is the time students must wait before feedback is available. We observed that 75 % of all JavaScript exercises are assessed automatically in less than 725 ms (figure 3.7). On average, TESTed spends 225 ms parsing and processing a TESTed-DSL test suite, with the remaining time used for generating test code (language-specific test harnesses), running test code and checking test results. Due to additional speedups in the testing framework itself, TESTed 2.0 is up to 2.8 times faster than TESTed 1.0 (both using JSON test suites).[8]

3.5. Results and contributions

In this section, we discuss TESTed-DSL and its impact on authoring test suites and task descriptions for programming exercises that support automated assessment. Apart from its application in authoring language-agnostic task descriptions, the three main contributions of TESTed-DSL for authoring test suites are in expressing how the requirements of programming exercises must be assessed. Its test suites:

1. share the exact same declarative structure and functionality across programming languages,
2. bridge between input/output testing (black-box, weakly typed) and unit testing (white-box, strongly typed), and
3. can express the test code in a language-agnostic way.

[8] `https://github.com/dodona-edu/universal-judge/pull/334`

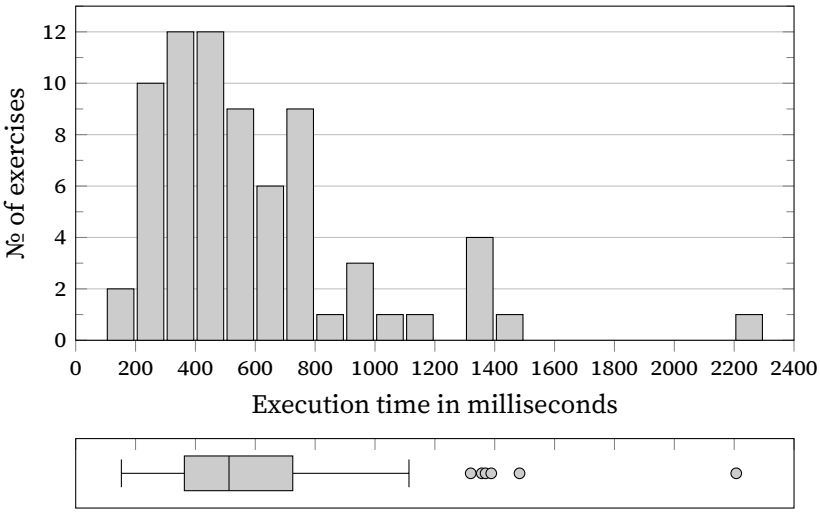


Figure 3.7. Histogram and box plot of the total runtime for automatically assessing the (correct) sample solutions with TESTed for all 72 JavaScript exercises. Half of the programming exercises have their submissions assessed in the range of 360 ms to 725 ms, with 75 % of the assessments taking less than 725 ms.

The first contribution might be of general interest to the broader software engineering community, but the last two contributions are especially relevant for computer science education. In the following subsections, we discuss the impact of each contribution.

For task descriptions, many of the same benefits apply as for test suites. Specifically, the application of the domain-specific language to task descriptions allows for programming language agnostic representations of code references: names of constructs (functions, classes, properties, etc.) and data types (e.g. lists, sets). Additionally, code fragments written in the abstract language can be automatically converted to code fragments of the target programming language.

3.5.1. Declarative structure

Most programming languages have one or more unit testing frameworks, whose structure and functionality are predominantly derived from Smalltalk's SUnit (Beck 1997). Collectively the latter are known as xUnit (Meszaros 2007). xUnit frameworks are code-driven as their test suites express both the structure and the behaviour of tests in the same programming

language as the software under test. Strong coupling to the language of the software under test is natural as behavioural tests must access its public interfaces.

In contrast, TESTed-DSL separates these two concerns by expressing the structure of test suites in a declarative way (in YAML). TESTed implements the functionality for processing this structural part of test suites in its core module, so only a single highly-optimized implementation is needed for this shared functionality of unit testing across programming languages. Functionality that remains language-dependent is the generation of test harnesses. But TESTed splits that across its core module generating language-agnostic harnesses and language-specific modules generating language-specific harnesses, to keep the language-specific modules as lightweight as possible and to share common functionality.

TESTed allows specifying language-specific test suites using TESTed-DSL that share the same structure across programming languages. TESTed simply has to copy the language-specific attributes when generating language-specific test harnesses. As an example, listing 3.6 shows a JavaScript-specific version of the test suite from listing 3.2. Test suites guiding automated assessment for the same exercise in other programming languages supported by TESTed, only differ in their representation for statements of the test scripts (`expression`, `statement`), expected values for strongly typed outputs (`return`, `exception`) and strongly typed arguments passed to custom oracles (`arguments`). The programming language in which these attributes (blue background) are expressed is specified with the `language` attribute (line 2) that is inherited across the hierarchy of the test suite.

A single TESTed-DSL test suite can also specify language-specific alternatives for multiple languages, as opposed to using a separate unit testing framework for each language, each with their custom version of the test suite. Listing 3.7 shows an example with Python and JavaScript alternatives for the same test expression. This gives access to language-specific features that are not (yet) supported in the abstract programming language of TESTed-DSL or are not (yet) supported by a language-specific module of TESTed. Here, the former is the case for passing anonymous functions as arguments to functions and converting strings to uppercase. This example also shows that TESTed-DSL test suites can mix language-specific (lines 2–4) and language-agnostic (line 5) sections.

```yaml
unit: "Cipher"
language: "javascript"
cases:
  - script:
      - statement: "const cipher = new Cipher('ABCD',
      ↪  '1AX3S1M2PYZ')"
      - expression: "cipher.grid"
        return:
          - ["-", "A", "X", "-"]
          - ["-", "-", "S", "-"]
          - ["M", "-", "-", "P"]
          - ["Y", "Z", "-", "-"]
      - expression: "cipher.map"
        return:
          "Y": "DA"
          "M": "CA"
          "P": "CD"
          "Z": "DB"
          "S": "BC"
          "A": "AB"
          "X": "AC"
      - expression: "cipher.encode('spam')"
        return: "BCCDABCA"
      - expression: "cipher.decode('BCCDABCA')"
        return: "SPAM"
      - expression: "cipher.encode('eggs')"
        exception: "invalid message"
      - expression: "cipher.decode('BCCDBACA')"
        exception: "invalid message"
  - script:
      # statements and expressions of the second script come here
```

Listing 3.6. JavaScript-specific test suite to validate correct behaviour of submissions that must define the class `Cipher`, where the shorthand was applied for test suites having a single unit. The JavaScript-specific attributes of this test suite are highlighted in blue.

```yaml
unit: 'Sort words'
expression:                          # language-specific
  python: "sort_words(['SPAM', 'eggs', 'bacon'], lambda word:
  ↪  word.upper())"
  javascript: 'sortWords(["SPAM", "eggs", "bacon"], word =>
  ↪  word.toUpperCase())'
return: ['bacon', 'eggs', 'SPAM'] # language-agnostic
```

Listing 3.7. TESTed-DSL test suite to validate correct behaviour of submissions that must either define the function `sort_words` in Python or define the function `sortWords` in JavaScript. The Python-specific sections of this test suite are marked in green and the JavaScript-specific sections in blue.

3.5.2. Combined input/output and unit testing

The examples in section 3.3 already show the flexibility of TESTed-DSL to specify test suites for both input/output exercises (listing 3.4) and exercises with fixed internal interfaces (listings 3.2 and 3.3). However, individual tests can also combine any mix of weakly and strongly typed input and output channels. For example, an expression calling a function with arguments, which reads from standard input, accesses main call arguments and environment variables, writes to standard output and error, and returns a value. In addition, test cases can combine black-box tests for the main call with a test script of white-box tests for statements and expressions that can access internal interfaces of the submission. This unification of test suites combining strongly/weakly typed and black-box/white-box approaches solves a long-standing problem in educational software testing.

Fonte et al. (2013) proposed Output Semantic-Similarity Language (OSSL) as a domain-specific language to serialize strongly typed data across weakly typed standard input and output. Although the formal specification of OSSL resembles the TESTed basic types supported by TESTed-DSL, no OSSL-parser was ever published and it was only conceived for use in language-agnostic input/output testing. For application in language-specific unit testing, TESTed added the extra layer of advanced types that are also supported by TESTed-DSL. Enstrom et al. (2011) suggest splitting programming exercises with complex tasks over multiple exercises, each testing a separate subtasks via input/output testing. Compared to the approach taken by TESTed-DSL, this feels like a poor man's version of white-box testing. It forces students to reveal internal interfaces via standard input and output, and at the same time asks authors to design and maintain alternative instances of an exercise. An accumulation of extra work, where what we actually seek are ways to reduce the work of authoring and solving programming exercises (Douce et al. 2005).

ACM-ICPC programming contests and derivatives are another area where input/output exercises are regularly used. These exercises often follow a specific style, with multiple test cases bundled in a single input stream. This style dates back from times where judges – as automated software testing frameworks are commonly called in the context of programming contests – were restricted to a single execution of the submission's main function. These platforms are heavily used in classrooms (Wasik et al. 2018; Zinovieva et al. 2021).

Although the ACM-ICPC style of testing can be expressed using TESTed-DSL test suites, it forces students to implement a main function that loops over test cases and bundles their output in a single output stream. This

approach of packing multiple test cases into a single test case further increases the black-box nature of testing. Additionally, it may propagate faults in the implementation of submissions as a failure for one test case to failures for successive test cases, making it harder to report feedback that easily discerns which individual test cases pass or fail. In programming contests, limiting feedback to reporting whether all test cases passed is not a concern.

However, in an educational context, we do strive for rich and fine-grained feedback, meaning an automated testing framework designed for contests is less than ideal. In TESTed-DSL, we essentially move the responsibility for processing multiple test cases from the student to the testing framework. This is useful for all students, but especially benefits those with limited programming experience.

3.5.3. Language-agnostic testing

TESTed-DSL's declarative structure to organize multiple test cases and describe individual test cases already enables authoring programming exercises whose submissions can be automatically assessed across programming languages. The abstract programming language means that test statements and strongly typed expected outputs can be specified once for all supported programming languages, eliminating repetition for each individual language that is a target for the programming exercise.

The need for language-agnostic software testing is relatively unique to programming exercises, where it is also highly relevant to support automated assessment. Beyond this educational context, we do not see many use cases for having a single specification to test multiple implementations that differ in programming language. Those use cases do exist (e.g. testing multiple implementations of the same standard), but they are often limited to input/output testing.

Besides online learning platforms supporting computer science courses in secondary and higher education, some platforms target self-learning, programming contests or recruitment (Hidalgo-Céspedes 2023). Examples include CodeWars[9], Edabit[10], LeetCode[11], CheckIO[12], Exercism[13] and CodingBat[14]. As far as we know, TESTed is the only existing frame-

[9] https://www.codewars.com/
[10] https://edabit.com/
[11] https://leetcode.com/
[12] https://checkio.org/
[13] https://exercism.org/
[14] https://codingbat.com/

work that bridges unit testing with language-agnostic testing. Most other frameworks are either restricted to input/output testing or rely on general-purpose unit testing frameworks. The input/output-based approach forces students to implement an input/output model in the main function and test oracles to process weakly typed data, and can only evaluate the behaviour of submissions as a whole.

Because general-purpose unit testing frameworks work with language-specific test suites, the alternative approach duplicates efforts in specifying the expected behaviour for a programming exercise to each target programming language of its submissions. This can be seen in how separate test suites are written for each target language in most programming platforms.

Only Exercism has a mechanism by which some exercises have a single specification in a generic track that is used to automatically derive language-specific instances.[15] This system shares its goals with TESTed-DSL. However, it is far less flexible and ergonomic for authoring programming exercises, as the generation step is needed each time.

3.6. Conclusions and future work

Version 2.0 of TESTed introduces TESTed-DSL as a domain-specific language for writing the test suites that underlie automated assessment, and for writing language-agnostic task descriptions. We have paid special attention to performance to ensure that TESTed-DSL has no additional overhead compared to JSON test suites (in fact, TESTed-DSL is 1.2 times faster than JSON test suites). TESTed-DSL itself, apart from its use in authoring language-agnostic task descriptions, has three main contributions: *i*) sharing the same declarative structure across programming languages, *ii*) bridging the gap between input/output testing and unit testing, and *iii*) allowing test code to be expressed in a language-agnostic way. The first contribution may be of general interest to the broader software engineering community, but the last two contributions are particularly relevant to computer science education.

Our goal is to further develop TESTed as an educational software testing framework for authoring different types of programming exercises across programming languages. This chapter has focused mainly on dynamic testing, but TESTed also performs compilation and linting as

[15] `https://github.com/exercism/problem-specifications`

generic types of static testing that are preconfigured in the language-specific modules. One area of future interest is the introduction of a language-agnostic interface for static code analysis.

We are currently investigating possible extensions to the abstract language of TESTed-DSL, such as operators for testing operator overloading, string conversion, comments, indexing sequences, indexing mappings, destructuring, object identity checking, and object equivalence checking. There is no need to conservatively restrict the abstract language to features supported by all or most programming languages, as TESTed automatically detects and reports features not supported by a specific programming language or not (yet) implemented by its language module. Support for additional test inputs (file descriptors, environment variables) and outputs (file descriptors, global scope) is in progress. We are also investigating native support for pretty printing of nested data structures to make it easier to detect differences between expected and actual return values, and data-driven tests (parameterized tests) to further improve the readability of test suites, support dynamic generation of test data and boost performance of running tests. Our future roadmap also includes internationalization of named submission interfaces, different ways to measure code coverage, and support for hidden units/test cases that are visible to teachers but remain invisible to students to avoid gaming – also known as programming to the test (Peveler et al. 2019).

Further enhancements and improvements of TESTed will be driven by educational practice, with the creation of new programming exercises and the conversion of existing exercises to TESTed-DSL being a major driver. Feel free to run TESTed as a standalone command line tool, integrate it into your online learning environments, and let us know about interesting use cases. We ourselves now routinely rely on TESTed to create new programming exercises and to gradually migrate existing exercises to port them to other programming languages and to benefit from the additional features that TESTed brings. We also switched to TESTed when training (secondary school) teachers how to create programming exercises with automated assessment for Dodona. First of all because it covers most common cases, is easy to use, and teachers can use the same framework for educational software testing, regardless of their target programming language(s). As an open-source project on GitHub, we welcome the sharing of unsupported exercise scenarios, bugs and feature requests documented as issues.[16] We also welcome additional language-specific modules to support new programming languages.[17]

[16] https://github.com/dodona-edu/universal-judge
[17] https://docs.dodona.be/en/references/tested/new-programming-language/

Part II.

Scratch

Chapter 4.

The Scratch programming environment

> *In teaching the computer how to think, children embark on an exploration about how they themselves think.*
>
> — Papert, *Mindstorms*

This chapter is a short introduction to Scratch, both the programming language and the environment. It also explains how the source code for Scratch is organized. The main goal is to familiarize the reader with the Scratch environment. Technical details are given in later chapters where relevant.

4.1. The Scratch environment

Scratch is a visual programming language and environment (Resnick, Maloney et al. 2009).[1] Visual programming languages let programmers construct programs by graphically manipulating program elements, rather than textually (Kelleher and Pausch 2005). A subset of visual programming languages are block-based languages, in which programs consist of blocks that are clicked together, not unlike puzzle pieces or Lego bricks (Weintrop and Wilensky 2015). Scratch falls into this category: it consists of a set of blocks with different shapes and colours. Development on it began in 2002, by the Lifelong Kindergarten research group at the MIT Media Lab. Scratch became publicly available in 2007 and has been developed by the Scratch Foundation since 2009. Its target audience is young learners, ages 8 to 16, although it is most commonly used for ages 10 to 14. It is a widely used programming language: the 2022 annual report of the Scratch Foundation (Scratch Foundation 2022) states that there are over 50 million users in the online Scratch community, with 120 million new projects created in 2022. The official statistics show

[1] `https://scratch.mit.edu`

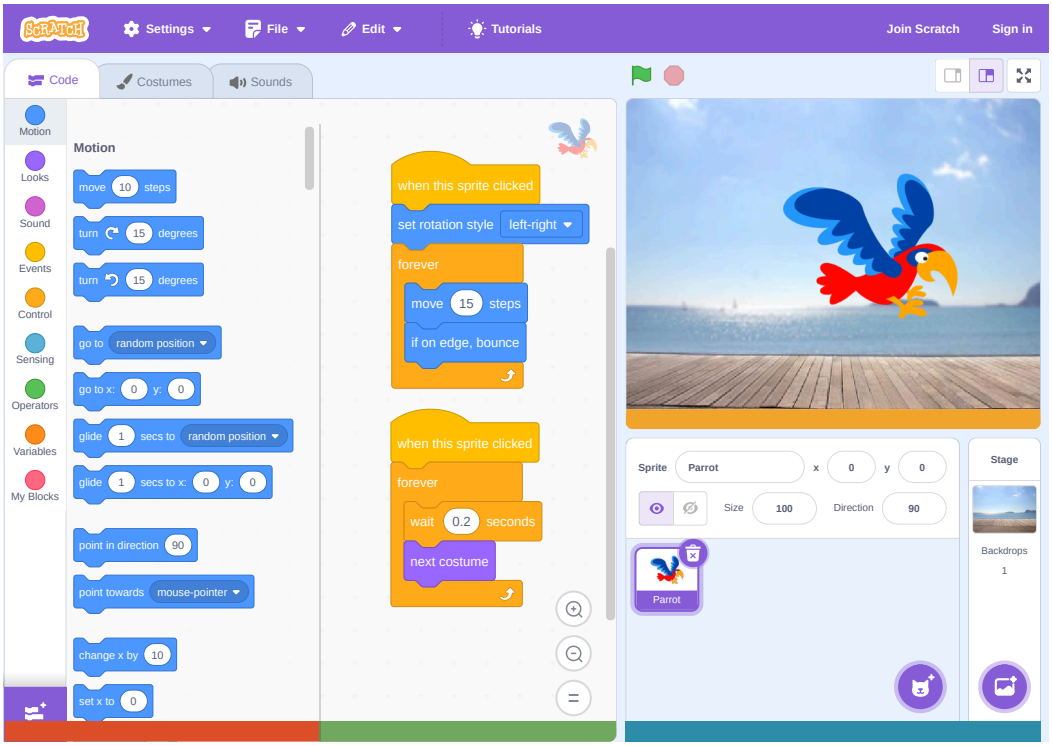


Figure 4.1. The Scratch environment running an example project. The toolbox with the available blocks is underlined in red, the workspace is underlined in green, the editors for the sprites and the stage are underlined in blue, and the canvas is underlined in orange.

that in May 2024, there were about 1.5 million active monthly users, with about 3.6 million new projects.[2] In April 2024, the Scratch team announced that the billionth project has been created.[3] The TIOBE index for May 2024 ranks it as the seventeenth most used programming language (TIOBE 2024).

The TIOBE index is disputed: it is based on the number of reported results in search engines (Bunce 2008, 2009; Sundarram 2022).

4.1.1. Using the environment and the blocks

Blocks can be dragged from the toolbox on the left side of the integrated development environment (figure 4.1) to the workspace in the middle and can be stacked together to form scripts. Blocks are categorized by

[2] https://scratch.mit.edu/statistics/
[3] https://twitter.com/scratch/status/1778814544682295394

their subject or goal: Motion, Looks, Sound, Events, Control, Sensing, Operators, Variables, and My Blocks (not counting extensions). Each category has its own colour, except for the blocks that handle lists, as these appear in the Variables category but have a different colour.

Each script starts with a hat block that defines when the script should execute. Scripts can be started as a result of a user action or when a certain criterion is met during the execution of a program, for example, when a clone is started or a message is broadcast. A common way to start scripts is to press the green flag ( ), which has two functions: it will first stop all running threads before starting new threads for all relevant hat blocks. The red stop button next to the green flag also stops all currently running scripts. The green flag button has another functionality: when there are active scripts, the button gets a darker background colour, even if those scripts were not started by the green flag.

Blocks (or parts of scripts) can also be run independently by clicking them. This illustrates that Scratch is always live: once a project has been loaded, the virtual machine is always running. Sprites can be moved or manipulated by the user at any time, even if scripts are running.

Each script is connected to a sprite. Sprites are objects that are drawn on the screen. The bottom right corner of the environment contains an editor for sprites and the background, which is a special sprite called Stage that is present in every Scratch project. All scripts corresponding to the selected sprite are shown in the workspace (middle). The canvas in the top right corner of the environment shows the execution of the project.

Besides the tab for the blocks (the "code"), there are also tabs for the costumes and sounds. The costumes are the visual representation of the sprite. While some blocks control which costume is used, the list of possible costumes must be prepared in this tab in advance. The sound tab is similar, but for sounds instead of costumes.

Finally, users can manage the sprites and stage at the bottom right. Sprites can be added and removed (even all sprites) by the user. Similarly, the "backdrop" of the stage (which acts as the costume for the stage) can be modified as well. Note that the stage cannot be removed.

4.1.2. Data types

Scratch has support for three basic data types: strings, booleans, and numbers (Maloney et al. 2010). A different block shape is used for booleans (a mix between rectangle and diamond) and strings/numbers

(round oval). Variables and reporter blocks (which are special blocks that result in some value) can only be inserted boolean slots if the shapes match. The string/number slots are less strict: if necessary, Scratch will coerce the data into the right type. Since the reporter blocks can also use operators, they fulfil the role of expressions.

In addition, Scratch also supports lists with their own set of blocks to manipulate them. Lists can only contain strings or numbers; booleans are cast to strings.

4.1.3. Sprites, the object model

Sprites are the Scratch equivalent to objects (Maloney et al. 2010). As all scripts belong to a particular sprite, almost all blocks only work for the current sprite. Apparently, an earlier version of Scratch had cross-sprite commands, but users found it confusing. Since Scratch lacks classes and inheritance, Maloney et al. call it an "object-based language".

The strict separation of code between sprites means there is a lot of work needed to make a set of sprites behave the same way. For example, a firework might need hundreds of sprites to represent the particles. Creating copies of the sprites by hand quickly becomes tedious. That is why Scratch has a clone feature: a "shadow" sprite is created that shares its code with the original sprite. While the code is shared, the execution is not: each clone can execute code independently. A clone is not visible in the sprite overview, only on the canvas.

Variables are normally also limited to the sprite that defined them and are not visible to other sprites. There is an exception: the variables of the stage are visible to all sprites and could thus be used for inter-sprite communication.

4.1.4. Inter-sprite communication

Broadcasts are the intended way to let sprites communicate with each other. There are other ways, such as variables (since variables in the stage are global). However, broadcasts remain the intended way to do inter-sprite communication. It is a one-to-many broadcasting system (Maloney et al. 2010): a broadcast (an arbitrary string) is sent globally and might trigger multiple scripts (even in different sprites).

4.1.5. Defining custom blocks with procedures

Scratch allows defining custom blocks, sometimes called procedures. This allows the user to define blocks that consist of other blocks, similar to procedures in other languages. Procedures in Scratch can have parameters that can take arguments, which are available as variables to the blocks of the procedure. Since the custom blocks are procedures and not functions, they do not have return values.

4.1.6. Concurrency and parallelism

Scratch is a highly concurrent language: every script is akin to a thread and can run concurrently to other scripts, even within the same sprite. The chosen concurrency model of Scratch has a few advantages but also disadvantages, which we discuss in section 7.3.1.

Due to JavaScript's single-threaded nature (the language in which Scratch is implemented), the actual execution of a Scratch program will not be in parallel.

4.2. Organization of the source code

The first public release of Scratch was in May 2007 as a desktop application (figure 4.2), written in Squeak, a Smalltalk implementation. This version already supported sharing projects by uploading them to the Scratch website. Scratch 2.0, released in May 2013, was a rewrite providing the first "online" version of Scratch (running in the browser). It used Adobe Flash for the online version and Adobe Air for the downloadable offline editor.

At the time, some users speculated that Scratch 3 was "rushed" since it was announced in 2017 that support for Flash would end in 2021. Work on Scratch 3, however, started in May 2016.

In January 2019, Scratch 3.0 was released. This version is a complete rewrite, using web technologies, such as JavaScript, HTML, and CSS. Scratch 3.0 is fully browser-based but still supports an offline editor using Electron.[4] Scratch 3.0 consists of a set of independent source code modules that work together to form the complete Scratch environment. An overview of the relationships between the modules is shown in figure 4.3. Each module is developed independently in a separate repository on GitHub.

[4] `https://scratch.mit.edu/download`

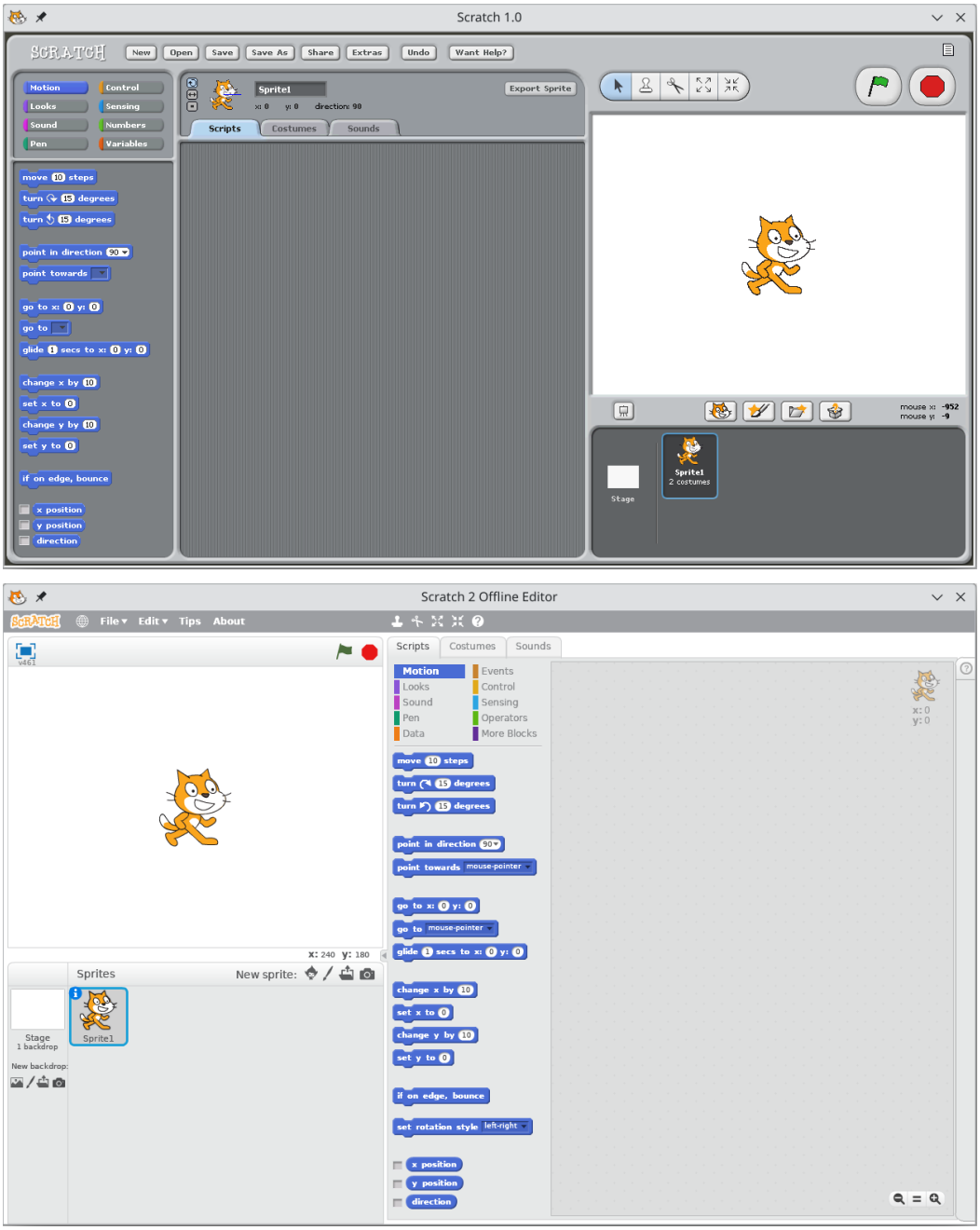


Figure 4.2. The Scratch 1.0 desktop application (top) and Scratch 2.0 offline editor (bottom).

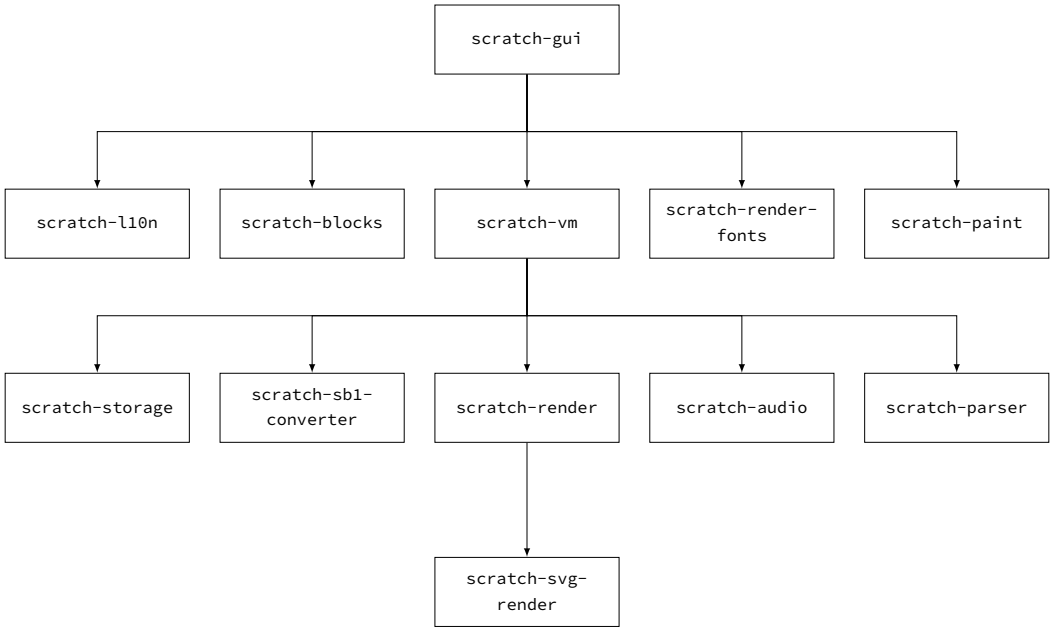


Figure 4.3. Overview of the dependencies between the Scratch repositories (using their repository name). The repositories that are purely for the hosted instance, such as the website, account system, and forum are left out. An arrow indicates a dependency. For example, the `scratch-gui` package has five dependencies.

The most important and relevant modules for this dissertation are:

Scratch Blocks A fork of Google's Blockly (Pasternak et al. 2017), a library for building block-based computing interfaces.[5]

Scratch Virtual Machine The runtime engine behind Scratch and responsible for running the projects created by the blocks.[6]

Scratch User Interface A React-based web application that consists of the Scratch programming environment. It uses and builds on the other components.[7]

Scratch Renderer A WebGL-based renderer, responsible for rendering the canvas.[8]

[5] `https://github.com/scratchfoundation/scratch-blocks`
[6] `https://github.com/scratchfoundation/scratch-vm`
[7] `https://github.com/scratchfoundation/scratch-gui`
[8] `https://github.com/scratchfoundation/scratch-render`

Chapter 5.

A testing framework for Scratch

> *There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs.*
>
> — Flon, *On research in structured programming*

Block-based languages like Scratch are popular tools for introducing computer science concepts to young learners (Bau et al. 2017; Zhang and Nouri 2019). The intuitive interface and focus on visuals make Scratch an engaging and accessible programming environment. Due to this visual nature, syntactical programming errors are eliminated and students can quickly create games, stories, or other projects. This approach gives students a lot of flexibility to express their creativity. However, functional errors will still occur (Zeller 2009). The process of teaching to code is often slowed down by the delay in providing feedback on and solutions to these errors.

When done manually, assessment of submissions for Scratch exercises is time-consuming and impractical, especially in large classes, where educators often lack the time to give individual feedback to each student. That is why it is important to equip students with tools that can provide immediate feedback and thus enhance their independent learning skills. In this chapter, we introduce Itch, a testing framework for Scratch that can act as an automated assessment tool (Douce et al. 2005).

Itch provides flexible testing capabilities: it supports both static and dynamic testing of Scratch projects. It also provides facilities to make common scenarios easy to test, allowing the educators to focus on testing the interesting parts of an exercise. We also reflect on Itch's use in an educational context and discuss what testing of Scratch exercises should look like. On the one hand, Scratch strives to allow maximal creative expression for students, while on the other hand, testable exercises need a well-defined goal and reasonable limits.

5.1. Related work

In this section, we look at the few existing testing frameworks for Scratch. Since they are few and far between, we also consider other tools with similar aims: helping students with Scratch. We begin by looking at the linter-like tools that perform static analysis.

One of the first ones is Hairball (Boe et al. 2013). Hairball analyses the blocks of a provided Scratch project, with multiple rules and analysers available. Dr. Scratch (Moreno-León and Robles 2015) allows analysing a Scratch project to provide various insights. For example, Dr. Scratch uses Hairball to assign a "computational thinking score" to the project, although the creators of Scratch are not a fan of this approach (Resnick and Rusk 2020), see section 5.6.3. A similar tool is Ninja Code Village (Ota et al. 2016).

QualityHound (Techapalokul and Tilevich 2017) is a linter that detects code smells like "duplicate code" or "broad variable scope". In total, twelve code smells are detected. LitterBox (Fraser et al. 2021) is a newer linter that shares some goals with QualityHound. For example, it will also detect code smells. LitterBox supports significantly more patterns and also finds potential bugs (Frädrich et al. 2020), by looking for code that seems suspicious (for example, comparing two literal values). Subsequent research has expanded LitterBox to find "code perfumes" (Obermüller et al. 2021), which provides positive feedback to students, for example, by noting good use of loops.

LitterBox can also translate code to LeILa, an intermediate language of the Bastet framework (Stahlbauer, Frädrich et al. 2020). Bastet uses this intermediate language to enable more traditional and advanced analysis and verification of Scratch programs. Examples of what the authors envision are automated test generation, data-flow analysis, unbounded model checking on predicate abstraction, and concolic testing.

Unfortunaly, we were not aware of ITCH when deciding to name our testing framework Itch.

ITCH (Johnson 2016) is the first automatic testing framework to the knowledge of the authors. It translates a limited subset of Scratch programs to Python. ITCH uses the Scratch say and ask functionality to perform input/output-based testing on Scratch projects. Of course, being limited to input/output, only a subset of the functionality of Scratch can be used.

Whisker (Stahlbauer, Kreis et al. 2019) is a fully automated testing framework for Scratch, and the most similar to Itch. While manually written test suites (also in JavaScript) are possible, Whisker focuses on automated testing (Deiner, Feldmeier et al. 2023). For example, Whisker (and

subsequent research) supports property-based testing, search-based testing (Deiner, Frädrich et al. 2020), and model-based testing (Gotz et al. 2022).

Finally, Scratch Testing Block (Nurue and Gray 2024) is a prototype for a Scratch extension that provides an "assert" block. It is in aim similar to Poke (section 5.7): providing testing facilities inside Scratch itself.

Besides Scratch, there are other block-based languages, for some of which testing frameworks also exist. Block-based languages, like MakeCode (Ball et al. 2019), that compile to another programming language (like Python or JavaScript) are less relevant here, as the testing frameworks often use the compiled version.

Snap*!* is another block-based language (Mönig and Harvey 2024), created in 2011 by a former member of the Scratch team. It has many features that are considered too advanced for Scratch, like higher order functions, prototype-based programming, nested sprites, and metaprogramming capabilities. SnapCheck is one testing framework for Snap*!*, inspired by Whisker (W. Wang et al. 2021). Another example is CodeMaster (Wangenheim et al. 2018), which also supports AppInventor (a way to create mobile apps using a block-based language). However, the actual testing is more similar to Dr. Scratch: it assigns scores based on a static analysis of the used blocks.

5.2. Introduction to Itch

Itch is an educational testing framework for Scratch exercises. Educators can write test suites (in JavaScript) that evaluate a submission to determine if the submission correctly implements the requirements in the problem statement of the exercise. A submission is the code (in this case the Scratch project) that students submit as the answer to a programming exercise.

Broadly, Itch provides two approaches to testing: static and dynamic tests. For the static tests, the project is not run: only the blocks are analysed. With static tests, the test suite can verify (discussed in section 5.3.2):

- If the students did not change some off-limit sprites.
- The metadata of the sprites, which sprites exist, their position, their size, their costume, etc.
- All kinds of checks on the blocks: only certain blocks were used, some blocks were not used, the blocks must match a particular pattern, the number of blocks, and so on.

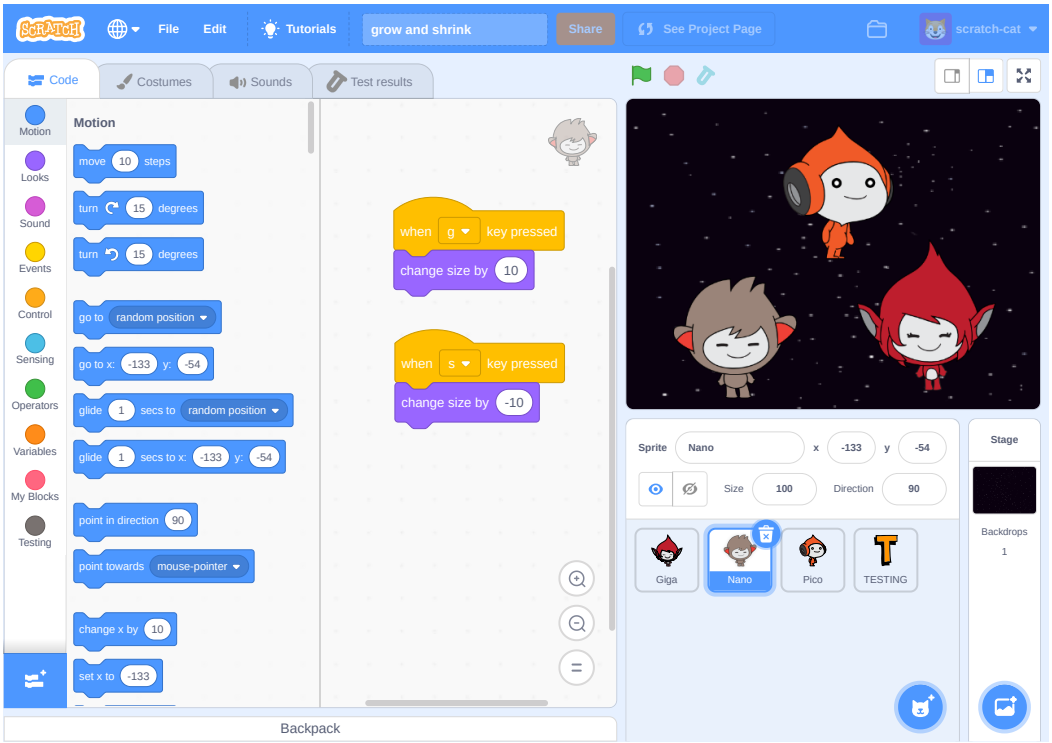


Figure 5.1. The *Grow and shrink* exercise. A correct implementation of the exercise for the sprite Nano is seen here.

However, the most flexible option is to use dynamic tests, which only look at behaviour, not at the implementation. Since these tests require the Scratch submissions to be run, Itch provides a scheduler that allows emulating user interaction (section 5.3.3) and other triggers. Itch can emulate pressing keys on the keyboard, moving the mouse, clicking the green flag, clicking on sprites, sending broadcasts, and setting variables. Additionally, Itch can wait some amount of time or for some requirement to be fulfilled (like a sprite moving to a certain position).

During this execution, Itch will save snapshots containing the complete state of the virtual machine. After the execution has finished, these snapshots are available for inspection in the log. Since the complete state of the virtual machine is saved throughout the execution of the project, almost anything can be checked. For example, checking if a sprite moved in response to a key press, or if a sprite moved up and down and switched costumes throughout the execution.

To illustrate Itch with a concrete example, we will use one exercise as a running example: the *Grow and shrink* exercise (figure 5.1). In this exercise, there are three sprites: each sprite must grow if the g key is pressed and must shrink if the s key is pressed. To test this exercise, we design a test suite that will verify the behaviour of the sprites (listing 5.1). This test suite will:

1. Save the existing size of the sprites.
2. Press the key that we are testing.
3. Verify that the new size of the sprite is larger or smaller than the previously saved size.
4. Save the new size as the existing size for the next test.

5.3. Test suites

Test suites for Itch are written in JavaScript. A test suite for Itch is split into three consecutive phases:

1. The **before execution** phase, which is run before the execution of the Scratch project.
2. The **during execution** phase, where the test suite controls the Scratch project and simulates user interaction.
3. The **after execution** phase, where tests are run on the log, which was collected during the previous phase.

All phases are optional: it is perfectly valid to not have a before execution, or only a before execution, depending on the types of tests that need to be run. They are implemented with "magic" functions: these have a fixed signature (name and arguments) and will be called in the relevant phases (listing 5.2).

Broadly, the tests can also be split according to their type: there are **static tests** and **dynamic tests**. Static tests do not require execution of the Scratch project. They are generally easier to write and faster to execute, but are severely limited in what they can test. Assessing whether a project uses a certain block (e.g. a loop block) somewhere in the project is typically done with static tests. Assessing whether an existing sprite (e.g. blocks that are provided in the starter project) was not modified is also done with static tests. Static tests can be done completely in the before execution phase.

```
function duringExecution(e) {
  const runtime = e.vm.runtime;
  // Save the original sizes of the sprites.
  const oldSize = {
    "Giga": runtime.getSpriteTargetByName("Giga").size,
    "Pico": runtime.getSpriteTargetByName("Pico").size,
    "Nano": runtime.getSpriteTargetByName("Nano").size
  }
  e.scheduler
    // Execute 4 events after each other.
   .forEach([1, 2, 3, 4], (prev) => {
     return prev.pressKey('g').log(() => {
       e.out.group("Test if sprites get bigger", () => {
         for (const sprite in oldSize) {
           const newSize =
           ↪  runtime.getSpriteTargetByName(sprite).size;
           e.out.test(`${sprite} got bigger`)
            .expect(newSize > oldSize[sprite])
            .toBe(true);
           // Save the new size as the old one.
           oldSize[sprite] = newSize;
         }
       })
     });
   }).forEach([1, 2, 3, 4], (prev) => {
    return prev.pressKey('s').log(() => {
      e.out.group("Test if sprites get smaller", () => {
        for (const sprite in oldSize) {
          const newSize =
          ↪  runtime.getSpriteTargetByName(sprite).size;
          e.out.test(`${sprite} got smaller`)
           .expect(newSize < oldSize[sprite])
           .toBe(true);
          oldSize[sprite] = newSize;
        }
      })
    });
  })
   .end();
}
```

Listing 5.1. The complete Itch test suite for the *Grow and shrink* exercise.

```
/** @param {Evaluation} e */
function beforeExecution(e) {
    // Tests go here
}

/** @param {Evaluation} e */
function duringExecution(e) {
    // Tests go here
}

/** @param {Evaluation} e */
function afterExecution(e) {
    // Tests go here
}
```

Listing 5.2. A skeleton of a test suite for Itch that shows the three phases. Each phase is implemented as a separate function that will be called at the appropriate time by Itch. The argument to these functions is an instance of the `Evaluation` class, which provides various methods to help with testing, such as the test structure, assertion functions, etc.

Checking more high-level goals, such as "Does the sprite move when clicked?", is more challenging with static tests. At least, without severely limiting the accepted solutions. For example, there are multiple ways to implement a sprite that moves when clicked, and a good behavioural test needs to accept all implementations. For these kinds of tests, dynamic tests can be used, which require the project to be executed. Dynamic tests often require both the during and the after execution phases. In the during execution phase, user interaction is simulated and the behaviour is captured. This captured behaviour can then be inspected in the after execution phase to see if the actual behaviour matches with the required behaviour.

Testing can also be done in the during execution phase, for example, after a user action has been performed. While this makes some test suites easier (like the one in listing 5.1), it does require attention to the parallel nature of Scratch. If multiple tests are run in parallel, the output can be garbled.

5.3.1. Structure of a test suite

A test suite consists of a hierarchical structure of groups and tests, which can be nested. A test is a check on some requirement or property of the exercise. It can be correct or wrong and can contain feedback for both

```
e.group.group('Tests for sprite A', () => {
  e.group
    .test('Sprite A does stuff right')
    .feedback({
      correct: 'Good job, sprite A does get stuff right!',
      wrong: 'Oh no, sprite A does not get it right.',
    })
   .expect('some value')
   .toBe('another value');
});
```

Listing 5.3. An example showing how the test suites are structured using groups and tests.

cases. Additionally, a test also has a name and can include additional information, such as value comparisons, messages, etc.

Structure is added by grouping the tests. Groups can be nested, so groups inside groups etc. are possible. While there is no hard limit, we recommend not going deeper than three levels in most cases (the user interface that displays the feedback can also impose limits). Groups are used for more than just structure. They also support a notion of visibility, with three modes:

Visible The group is expanded (the group and all its children are visible).

Hidden The group is completely hidden unless one of the tests in the group fails.

Summary The group is collapsed by default, and a summary is shown unless one of the tests fails (in which case the group expands).

Groups and tests in the JavaScript test suites are inspired by the Jest testing framework.[1] The two relevant methods, `group` and `test` are available on the `Evaluation.group` property passed to the test suite. Listing 5.3 shows an example of this, containing a single test. The double `group.group` is not a typo but needed for backward compatibility with older test suites. The test, as written in the example, will always fail, since it expects the string `"some value"` to be equal to `"another value"`. The `expect`/`toBe` notation specifically will be familiar to Jest users.

Maintainging backward compatibility is one of the downsides of using software in production.

Groups can be given just a name, as in the example, but it is also possible to provide more structured data. For example, it is possible to link a group to a certain sprite. This information is also provided in the generated feedback, which means the platform responsible for showing the feedback can show an image of the sprite with the group.

[1] `https://jestjs.io/`

More details on how this structure is reflected in the generated feedback can be found in section 5.5. Other possible metadata includes the visibility, a summary (used if the group's visibility is set to summary), some tags (which allows tagging groups with arbitrary strings), and an option to ignore wrong tests. This last option means that if a test fails, it will be ignored completely and will not be outputted in the generated feedback. This can be useful in cases where there are multiple possibilities: this allows trying each possibility until a passed test is found.

5.3.2. Before execution

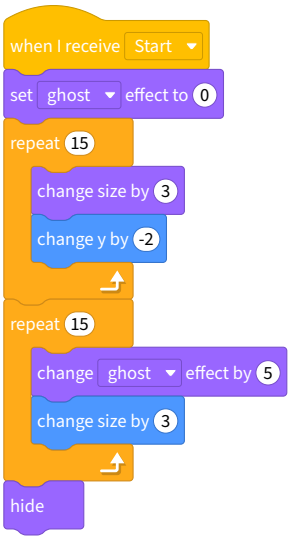
The before execution phase allows for executing static tests. Itch provides access to representations of two projects: the submission, created by the students, and the starter project, which is the project the students started with.

Access to both projects provides an easy way to assess whether the students modified some sprites or blocks in their submission. Itch provides helpers to ensure that students only modified certain sprites, or only certain code within sprites. While this does limit the students in their ability to creatively modify the project, it makes behavioural tests for complex projects easier. By limiting where the students are allowed to modify blocks, later tests can rely on certain functionality or sprites being available and working. For example, if the project contains blocks that check if a sprite touches another sprite, these can be relied on.

This functionality of checking a set of predefined blocks and sprites is exposed with the function `Itch.checkPredefinedBlocks` to test suite authors. For example:

```
1  Itch.checkPredefinedBlocks({
2    spriteConfig: {
3      SpriteA: script(whenIReceive('Start'), setEffectTo('ghost',
       ↪  0)),
4      SpriteB: {
5        pattern: script(whenIReceive('Start'), setEffectTo('ghost',
         ↪  0)),
6        allowedBlocks: [forever()],
7        allowAdditionalScripts: true,
8      },
9    },
10   debug: false,
11 }, e);
```

The example above contains tests for two sprites. All other sprites must be unchanged.

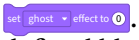


(a) A script in Scratch.

```
1  script(
2    whenIReceive('Start'),
3    setEffectTo('ghost', 0),
4    repeat(15, script(
5      changeSizeBy(3),
6      changeYBy(-2)
7    )),
8    repeat(20,script(
9      changeEffectBy('ghost', 5),
10     changeSizeBy(3)
11   )),
12   hide(),
13 );
```

(b) The equivalent in JavaScript.

Listing 5.4. An example of how a Scratch program can be represented using the abstractions provided by Itch.

For the sprite `SpriteA`, students are allowed to change, add, or remove blocks in the script starting with the two blocks  and  . Students are allowed to change blocks after these two predefined blocks.

The second sprite, `SpriteB`, allows modifications to scripts starting with the same blocks, but it uses the full version of the config object. With the full version, it is also possible to specify if additional scripts are allowed and limit the blocks they can use. In this example, only the  block can be used (so this is not a useful test). Finally, additional scripts (thus new ones created by the students) are also allowed. These are free of restrictions: the check for allowed blocks does not apply.

Itch also includes an abstraction to represent Scratch blocks in JavaScript, as used in the example above. For each Scratch block, a corresponding function exists (see an example in listing 5.4).

The functions representing blocks can also be used to construct block patterns. Two additional functions are provided for patterns. The first is `anything()`, which can be used in any location (as a block or value) and matches any block or value. The reverse, `nothing()`, matches no

```
/** @param {Evaluation} e */
function duringExecution(e) {
  e.scheduler
    .greenFlag(true)
    .wait(800)
    .pressKey('s')
    .end();
}
```

Listing 5.5. An example of the during execution phase where the scheduler is used to first press the green flag, wait 800 ms, press the s key, and finally end execution.

block or value. It can be useful to ensure that a script terminates (i.e. that there are no more blocks afterwards). Additionally, an array of blocks or patterns can also be used in most places. This represents a choice: it will match any of the patterns in the array.

For example, consider the following pattern:

```
repeat([15, 30], script(changeSizeBy(any()), never()))
```

It will match a 

block that repeats 15 or 30 times, with a body with exactly one block, 

, whose argument can be anything.

Since scripts form a tree of blocks, there are also helpers to match and test against a script of blocks. This supports error messages for each block (meaning they each show up as a failed assertion in the feedback).

5.3.3. During execution

The during execution phase is actually run just before the project is executed by Itch. The main purpose of this phase is to use the `e.scheduler` (an instance of the `Scheduler` class) to schedule the execution of the project. Using the scheduler, the test suite must specify how the project should be executed. This includes starting execution, stopping execution, manipulating the virtual machine, and simulating user interaction, like clicking, key presses, input, and so on. Listing 5.5 shows a minimal example of a schedule where the green flag is pressed, 800 ms must pass, and finally the s key is pressed.

The scheduler receives a set of actions to perform. Each next action is executed after the previous one has been scheduled or completed. Because Scratch is a highly concurrent language, the scheduler also supports this.

First, most actions support a synchronous and asynchronous variant. In the asynchronous variant, the action is executed and immediately finished. For example, the action to press the green flag is almost instant: the green flag is pressed in the virtual machine, and the action is complete.

The synchronous actions will only finish after all activated scripts in Scratch have terminated. For example, the action to press the green flag will wait until all scripts with the hat block `when flag clicked` are done executing. Of course, there are scenarios where this is not possible. If one of the scripts contains an infinite loop, the next scheduled action would never be performed, as the script will never end.

Since the project that is run comes from students with unknown code, synchronous actions also support a timeout. After this time has passed, a failed assertion will be added to the generated feedback, and the scheduler will continue.

Every action in the scheduler returns the last scheduled action: this return value can be used to schedule the next action after the previous one. To create a non-linear schedule (e.g. multiple actions are performed simultaneously), there are a few options. First, the return value of one of the previous actions can be used multiple times to schedule new actions. All of these will be run in parallel.

Another option is to use the method `forEach`: this is an implementation of the "fold" function on the scheduler events. By deciding what action is used as the accumulator, either a linear (by returning the new action) or non-linear schedule (by returning the existing action) can be created. Figure 5.2 shows an example of this. By returning the new action, each action L1-4 will be performed in succession. By returning the original action, the N1-4 actions will all be scheduled at the same time. Since both L1 and N1 are scheduled on the same starting action S, they will also run in parallel.

While most actions are equivalent to their counter-parts in Scratch (like clicking a sprite, pressing a key, etc.), the wait action has more features. In addition to waiting a set amount of time (like in the example), the wait action can also wait on the fulfillment of a certain condition. The wait condition can be either waiting on a certain broadcast being sent, or a sprite fulfilling some condition. The possible sprite conditions are moving, reaching a certain position, touching another sprite, no longer touching another sprite, touching the edge, and touching the mouse. These conditions are added as needed, so with use, we envision more conditions being added.

```
const s = e.scheduler.wait(500);

// Schedule actions L1-4
s.forEach([1, 2, 3, 4], (p) => {
    return p.wait(500);
});

// Schedule actions N1-4
s.forEach([1, 2, 3, 4], (p) => {
    p.wait(500);
    return p;
});
```

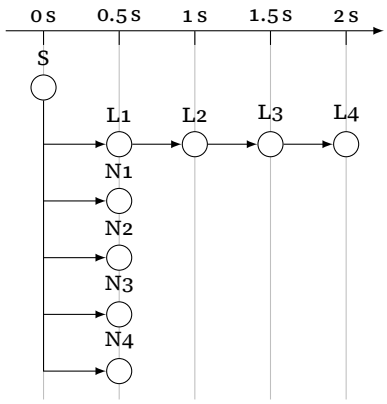


Figure 5.2. The code for the scheduler (left) and the resulting schedule of actions (right). The start of an action is shown with a circle. After the start action S, the L and N actions are scheduled at the same time. However, the L actions return the new action in the function, meaning L1-4 will run sequentially. The code for the N actions returns the original action, meaning N1-4 will run concurrently. Since both N1 and L1 are scheduled on the same action S, the actions L1 and N1-4 will all run concurrently.

A final special action is the `log` action. This action saves the current state in the log (section 5.3.4) and executes a custom function at that time. This action is intended to mark certain events in the log but can also be used to execute a test during the execution of the project, instead of beforehand or afterwards. While the test is executed during the execution, it also has access to snapshots and events that have already taken place.

5.3.4. After execution

Itch uses the scheduler from the during execution phase to execute the submission. During the execution, a log is constructed of the execution. This is done by hand (using the `log` scheduler action) and automatically at interesting points. The log consists of snapshots (which are taken every time something changes in the virtual machine) and events (which denote interesting snapshots). For example, every action in the scheduler is saved as an event, meaning there is a snapshot before the action and after the action has been completed.

Returning to the *Grow and shrink* exercise, listing 5.6 shows an alternative test suite (compared to listing 5.1). In the during execution phase, only the key presses are scheduled. Afterwards, in the after execution phase,

```
function duringExecution(e) {
  // Schedule pressing each key four times in succession.
  e.scheduler
   .forEach([1, 2, 3, 4], (prev) => prev.pressKey('g'))
   .forEach([1, 2, 3, 4], (prev) => prev.pressKey('s'))
   .end();
}

function afterExecution(e) {
  const sprites = ["Giga", "Pico", "Nano"];

  // Get the events for the g key presses.
  const gPresses = e.log.events.filter((e) => e.type === 'key' &&
  ↪  e.data.key === 'g');
  for (const event of gPresses) {
    e.out.group("Test if sprites get bigger", () => {
      for (const name of sprites) {
        // Get the sprite from the snapshot before the key press.
        const before = event.previous.sprite(name);
        // Get the sprite from the snapshot after the key press.
        const after = event.next.sprite(name);

        e.out.test(`${name} got bigger`)
         .expect(after.size > before.size)
         .toBe(true);
      }
    });
  }

  const sPresses = e.log.events.filter((e) => e.type === 'key' &&
  ↪  e.data.key === 's');
  for (const event of sPresses) {
    e.out.group("Test if sprites get smaller", () => {
      for (const name of sprites) {
        const before = event.previous.sprite(name);
        const after = event.next.sprite(name);

        e.out.test(`${name} got smaller`)
         .expect(after.size < before.size)
         .toBe(true);
      }
    });
  }
}
```

Listing 5.6. Alternative test suite for the *Grow and shrink* exercise. In contrast to listing 5.1, this test suite only schedules the key presses in the during execution phase. All tests are performed in the after execution phase using the log.

the log is used to perform the tests. For each key press, the corresponding event in the log is looked up. Each of these events provides before and after snapshots, which are then used to determine if sprites correctly change in size.

5.4. Evaluating projects

The evaluation of a submission goes through the following process:

1. The submission, the starter project, and the test suite are made available.
2. Itch loads both projects, and runs the before execution phase. While the starter project is available in all phases, it is only intended to be used in the before execution phase.
3. If the before execution phase does not result in an error, Itch initializes the virtual machine, inserts hooks for the log, and loads the submission into the virtual machine.
4. The during execution phase is run, and the scheduled actions are read from the test suite. The during execution phase is used to create a schedule of user actions using the scheduler. It does not run the schedule itself.
5. Itch starts the virtual machine and executes the scheduled actions. While executing, the logs are captured.
6. If the execution does not result in an error, Itch shuts down the virtual machine, and runs the after execution phase.
7. All feedback is sent to the output. The output is a callback function that can be provided when starting Itch, or the default function is used, which prints the feedback on standard output.

Scratch 3.0 is built using browser technologies and has to be run in the browser. While the virtual machine is pure JavaScript and could thus run without a browser, the renderer is not: it uses the HTML canvas and WebGL technologies. The renderer is used to calculate things like sprite collisions and checking if sprites touch certain colours. While it is theoretically possible to create a renderer that does not use WebGL, this would be a big undertaking and imply a big maintenance burden. The re-implemented renderer will have to be kept up to date with the upstream one and replicate all behaviour exactly.

```
export interface EvalConfig {
  /** The submission sb3 data. */
  submission: ArrayBuffer;
  /** The starter project sb3 file. */
  template: ArrayBuffer;
  /** If the output should be partial or full. */
  fullFormat: boolean;
  /* The canvas for the renderer. */
  canvas: HTMLCanvasElement;
  /** The test suite to use. */
  testplan: string | TestplanSource;
  /** Callback for the results. */
  callback: OutputHandler;
  /** The language of the exercise. */
  language: string;
}

/** Run the judge. */
export async function run(config: EvalConfig): Promise<void>;
```

Listing 5.7. The exposed interface to run Itch. It consists of one function and a configuration object.

Itch consists of two JavaScript packages, which is a reflection of the two ways to run Itch:

- As a library in the browser. This is useful for contexts where there already is a browser, e.g. running Itch along the Scratch environment on the device of the student for client-side testing. This is implemented by the first JavaScript package, the core package.
- As a command line tool. This is for cases where Itch runs as a service in the backend, e.g. to check submissions after students are done. This is implemented in the second JavaScript package, the runner package. This package wraps around the core package to launch the browser instance and load the various dependencies (it thus runs the core package, which is where it gets its name).

5.4.1. Running Itch as a library

When running Itch as a library, the core package must be loaded into the browser, in addition to having the Scratch dependencies (the virtual machine, etc.) available. The exposed interface to run Itch is limited to one function and a configuration object (listing 5.7). Most of the options are self-explanatory, and the output format option is explained in section 5.5.

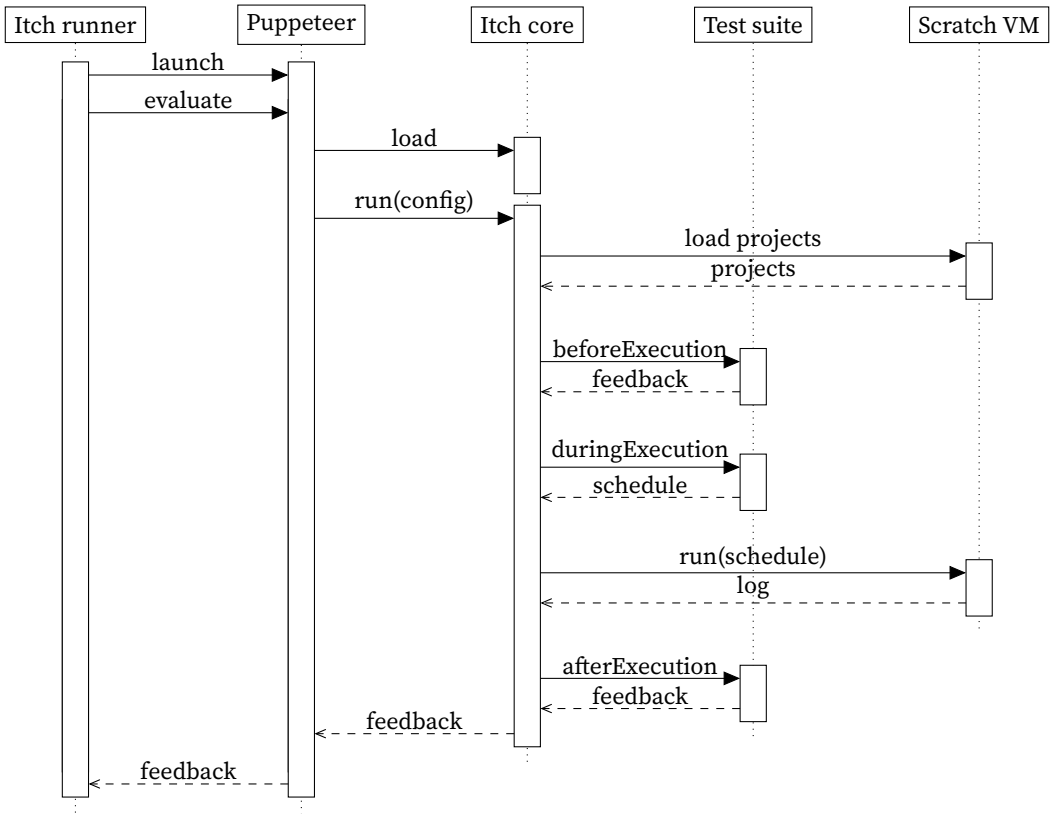


Figure 5.3. Sequence diagram showing the process of evaluating a project with Itch. When run as a library, there is no Itch core or Puppeteer.

5.4.2. Running Itch as a command line tool

When there is no existing browser instance available, Itch provides a command line interface. In this mode, Itch will run a headless browser, load the projects and test suites, run the judge, and finally collect the results from the browser.

A headless browser is a full browser, but without user interface.

Figure 5.3 shows the complete process. First, the runner package launches a Puppeteer instance (the headless browser). When ready, the various dependencies are loaded into the Puppeteer instance (these are the Itch core package, the Scratch dependencies, the test suite, the submission, and the starter project). The runner package will then run a special script that calls Itch as it would be used as a library: the `run(config)` function is called. Then, the same process happens as when using Itch

as a library. First, the projects (submission and starter) are parsed. The phases are then executed, with the before and during execution phase going first. The during execution phase results in a schedule, which is then run. The virtual machine is started and runs the projects with the scheduled actions. Finally, the after execution phase is run, with the log from the previous phase.

5.4.3. Performance considerations

Reducing the time students spend waiting on feedback is essential for providing a good user experience with software testing, especially in educational software testing (Sarsa et al. 2022). One difficulty in providing fast feedback on Scratch code is that blocks that wait are commonly used. However, these wait times pose a hard lower limit on the duration of the evaluation of a Scratch project. For example, if a project has a wait block of 2 seconds, the evaluation time of that project can never be less than 2 seconds (and will be higher in practice, since the whole evaluation process also adds overhead). Such wait blocks can quickly add up.

As a solution for this problem, Itch provides test suites with the ability to set an "acceleration factor": this factor indicates the speedup for all time-based data, both in the project and in the test suite. For example, if the acceleration factor is 2, the wait time in wait blocks will be halved, as will wait times in the scheduler. The acceleration factor can, however, be flaky with high values: we do not recommend a factor higher than 5. If it is expected that, for example, wait blocks are used for synchronization of two sprites, the acceleration factor can cause issues, as it does not speed up general execution. For example, if a wait block has a value timed to wait as long as the execution of another script with five blocks takes, the acceleration factor will cause the wait block to stop waiting early.

Itch also provides the test suite with the possibility to enable Scratch's built-in turbo mode. In this mode, execution is done faster (how much depends on the project). Together with the acceleration factor, these techniques can significantly speed up the evaluation.

5.5. Format of the generated feedback

The format of the generated feedback is in structure similar to the structure of the tests in the test suite (section 2.6.1). The three levels of the feedback are:

```
{"command": "start-judgement", "version": 2}
{"command": "start-group", "name": "Check on existing code",
↪  "visibility": "summary"}
{"command": "start-group", "name": "Stage", "sprite": "Stage",
↪  "visibility": "summary"}
{"command": "start-test", "name": "Sprite exists"}
{"command": "close-test", "feedback": "The sprite exists",
↪  "status": "correct"}
{"command": "close-group"}
{"command": "close-group"}
{"command": "close-judgement"}
```

Listing 5.8. Example of the output generated by Itch for a test suite with two nested groups, with one test. Note the similarity to listing 2.4.

1. **Judgement**: the top-level object of the feedback.
2. **Group**: contains one or more tests or subgroups (equivalent to a group from the test suite). Each group has the same options as in the test suite (e.g. for its visibility, a linked sprite).
3. **Test**: one condition or requirement that is evaluated (equivalent to a test from the test suite).

Itch uses a callback function to process feedback and has two modes. In partial mode, the callback is called whenever feedback is available. Full mode means all feedback is collected and the callback is called once with the collected feedback.

When running Itch on the command line, a default callback function is used. This default will send all feedback to standard output in a format similar to the Dodona feedback format used by TESTed (section 2.8.3). The feedback is printed as newline-delimited JSON, meaning JSON objects are separated by a newline, and each line is a valid JSON object. Listing 5.8 gives an example of this format. In partial mode, the structure of the feedback is indicated by commands, with `start` commands to begin a new level in the hierarchy and `close` commands to finish a level. The full mode is similar, but there is only a single big JSON object that represents a nested tree of the same feedback.

5.6. Itch in practice

In this section, we discuss the use of Itch in educational practice and discuss insights from creating test suites for Scratch exercises.

5.6.1. Capabilities of the testing framework

The initial development of Itch used a set of 13 Scratch exercises from the 2017 edition of the Flemish Programming Contest (*Vlaamse Programmeerwedstrijd*). The Scratch exercises were used in a special category for students aged 10 to 12. This category was a one-time event organized by the local organizer as a side-event for the Programming Contest, which normally focuses on text-based programming languages. It is worth mentioning that this event used human judges to assess whether the children had successfully completed an assignment. An example of one such exercise is given in appendix A. This was the case as there were no Scratch testing frameworks at the time that could automate this assessment: as a result, the exercises were not created with automated assessment in mind.

These 13 exercises were used as a reference for the initial feature set and capabilities of Itch. The exercises can be classified into five groups, depending on their contents:

Speaking Sprites speak, think, or ask questions (five exercises).

Moving Sprites move around (three exercises).

Costumes Sprites change costumes, often to emulate animation (three exercises).

Games Interactive games that require user input (one exercise).

Drawing The *Pen* extension is used to draw on the canvas (three exercises).

Note that the exercises do not sum up to 13: two exercises are both in the moving and costumes category, since they do both.

For 12 of the 13 exercises, an Itch test suite was created that tested the submissions sufficiently that it would have been usable in the programming contest. The remaining exercise was not testable due to its open-ended nature. The task description is, in summary, “draw a house”. Which leads to a vast variety of houses as solutions. Besides introducing some computer vision framework to detect houses (which would be technically possible in Itch), this is not easily testable.

These exercises were all tested with dynamic tests, meaning they test the behaviour of the submission. The exercises of CodeCosmos (which are discussed in section 5.6.2) are often longer interactive games. Due to their complex nature, static testing was sometimes needed (which is discussed in section 5.6.4).

5.6.2. Itch in educational practice

The use of Itch in educational practice has done mainly with our commercial partner, CodeCosmos. CodeCosmos provides the platform in which Itch is integrated. The platform provides a variety of organizational tools, like classroom management. It has its own hosted instance of the Scratch environment (similar to the MIT-hosted instance). This instance has been modified to integrate with Itch.

On the platform, Itch is run server-side by a dedicated service. The choice for server-side testing versus client-side testing was mainly motivated by making sure that students could not cheat by modifying the test suites or the results. The platform will periodically (or when a button is pressed) send the current version of the submission to the server. The backend will then look up the test suite and starter project and send these to the Itch service. After testing has been completed, the results are returned to the platform where they are displayed to the students in an additional, custom tab in the Scratch environment.

There are two different ways in which the CodeCosmos programming exercises are used. The first is as part of a "teaching pack", which is bought and used by schools. These packs implement the "attainment descriptors", which are documents that describe the learning objectives of the lessons and what skills students should possess after taking the courses. Here, the teaching pack provides a full learning path for the students, with suitable exercises at each stage of the process. The second way is as an extracurricular activity, where students either follow weekly lessons or join a "coding camp" for a few weeks in the summer. In both cases, many of the same exercises are used, but the context in which they are used differs.

Called eindtermen in Dutch.

In total, there are 139 exercises with an Itch test suite. In the period from March 3rd, 2023 until March 26th, 2024 (328 days), Itch evaluated 28 144 submissions. Of these submissions, 2713 (9.64 %) are evaluated as being correct, which means 90.36 % of evaluated submissions are incorrect. This is in line with our expectations: we allow students to submit as much as they want (and even automatically submit every so often) until the submission is correct. This also allows for collecting information about the trajectory of students while solving an exercise but also provides more data for analysing the growth of students throughout a course. Providing insights from this data to educators is one area of future work that is planned by CodeCosmos.

The submissions were made by 6496 unique users, with an average of 4.33 submissions per user. The submissions have an average test count

of 158, (the exercises have a number of tests that range from a minimum of 2 and a maximum of 1292).

Most of these exercises were conceived before introducing Itch, meaning Itch is capable of evaluating exercises as typically used by CodeCosmos. The biggest change we made to the exercises was splitting the levels of the exercises into different Scratch projects. However, coming up with a test suite for all of these exercises was a challenge, which is discussed next.

5.6.3. How to assess Scratch projects

The creators of Scratch have a well-known vision on how they intend Scratch to be used. The Scratch team prefers not to use assignment-based learning (Resnick and Rusk 2020). As a specific example of this, they particularly do not support strict static tests on exercises. For example, Resnick and Rusk state:

> Too often, researchers and educators are adopting automated assessment tools that evaluate student programming projects only by analysing the code, without considering the project goals, content, design, interface, usability, or documentation. For example, many are using an online Scratch assessment tool that gives students a "computational thinking score" based on the assumption that code with more types of programming blocks is an indication of more advanced computational thinking. This form of assessment doesn't take into consideration what the student's program is intended to do, how well it accomplishes the student's goals, whether the code works as intended, whether people are able to interact with it, or how the student's thinking develops over a series of projects.

They are probably talking about Dr. Scratch here.

At the same time, using linters or other static analysis tools to detect code smells is considered beneficial for students (Hermans and Aivaloglou 2016). However, in Itch, static analysis can also be used for automatic assessment by writing static tests. While we share the sentiment of Resnick and Rusk, our experience with Scratch exercises and our collaboration with CodeCosmos has shown that such tests are often used, despite their disadvantages. There are a few factors that contribute to this:

- Exercises (especially in Scratch) that were not created with testability in mind are hard to properly test without resorting to static analysis of the code.

- Scratch projects, especially complex ones, are technically challenging to test within an acceptable time frame. We cannot spend minutes evaluating a single submission, as feedback must be provided quickly to students (Sarsa et al. 2022).
- Educators want simple metrics to easily see if students have mastered some concept. For example, determining if a student has mastered the concepts of loops (or repetition in general) is challenging to do automatically. With a large number of students and time constraints, educators fall back on analysing blocks as a proxy for mastery (Combéfis 2022).

This does not mean that static tests are of no use: with good and thoughtful static tests, an educator (and student) can be fairly certain that a submission marked as correct is effectively correct. This can guide the educator to spend more time on submissions where their attention is warranted. Similarly, a submission marked as wrong can also help pinpoint the problem to the student, without intervention from the educator.

However, as so eloquently stated by Edsger Dijkstra, "Testing shows the presence, not the absence of bugs". Students might have solved the exercise in creative or innovative ways, which static tests fail to detect. Similarly, they might have fooled the static tests into marking the submission as correct, even though it is not. Therefore, we do not recommend using static tests to automatically grade submissions, at least not without verifying the results.

5.6.4. Creating test suites for Scratch exercises

Itch does not use the same programming language for the test suites as the programming language of a submission: test suites for Scratch exercises are written in JavaScript. This has advantages, as JavaScript is a general-purpose programming language, while Scratch is much more limited, but it also has disadvantages.

Doing the same in Scratch would be much less ergonomic then doing it in JavaScript.

It is not uncommon for educators that use or create Scratch exercises to have no formal computer science background, nor experience with programming besides Scratch (H. Kim et al. 2012; Oliveira et al. 2019). This is also the case at CodeCosmos, where teachers with an educational background design and create the exercises. For the existing exercises, the educators designed the exercise and implemented the Scratch parts, while others (like the author of this dissertation) implemented the test suites. However, it is our experience that authoring exercises is faster if

the people designing the exercise have knowledge of the testing framework and implement the test suite themselves. In section 5.7, we explore an experiment showing what a testing framework for Scratch with test suites in Scratch may look like.

5.7. Writing test suites in Scratch

As mentioned in the previous subsection, creating test suites in JavaScript for Scratch exercises can be challenging for educators without computer science background or experience. To fill this gap, we have developed Poke: a prototype of a testing framework for Scratch implemented in Scratch itself. This means that test suites are written with Scratch blocks, tests are executed in the Scratch environment, and the results are also shown in the Scratch environment.

For this prototype, our focus was twofold: "Is it possible to create a testing framework for Scratch in Scratch?" and "If possible, which testing facilities should be present?". The answer to the first question is yes, and the remainder of this section discusses the details of Poke, along with answers for the second question. These considerations include what blocks to provide, how tests should be run, etc.

5.7.1. Introduction to Poke

The user-facing part of the Poke software testing framework consists of three parts (figure 5.4):

- An additional button next to the green flag and stop button, which will run the tests.
- An extension to provide a set of blocks to write tests with.
- An additional tab in the Scratch environment that shows the test results once run.

5.7.2. The Poke extension

Poke provides five categories of blocks: a hat block (which starts the tests if the button is clicked), feedback blocks, user interaction blocks, observation blocks, and blocks to execute code in another sprite.

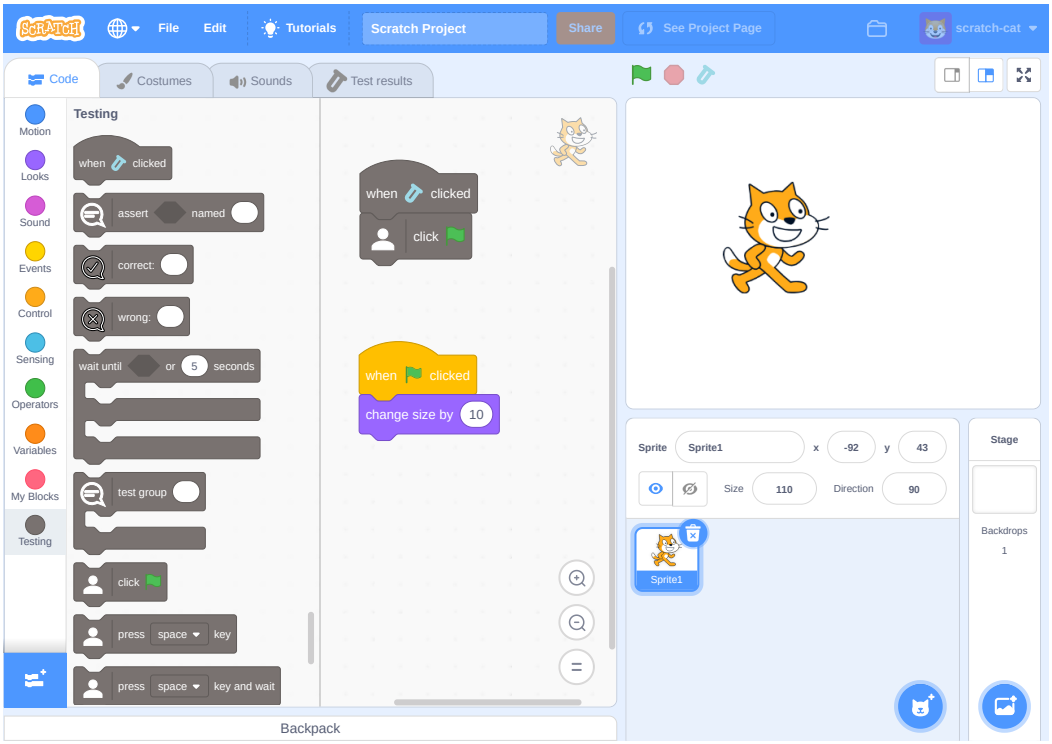
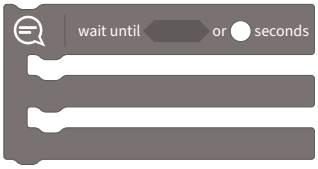


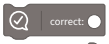
Figure 5.4. The Scratch environment with the added Poke elements: the additional button next to green flag and stop buttons to start the tests, the additional tab to show test results, and the extension that provides the Poke blocks.

Feedback blocks

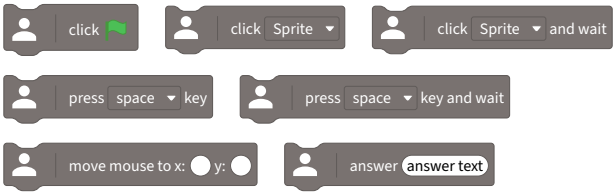
The structure of a test suite in Poke is identical to the structure in Itch (section 5.3.1): a test suite consists of groups, which can contain tests or other groups. The block for creating groups is a C block that takes the name of the group as an argument. The body of the block outputs its test results in the group.

For generating test results, there are three blocks, the main of which is the assertion block. It takes a boolean value and a name. The name is shown in the output, together with a status depending on the value of the boolean block. There are also the correct and wrong blocks, which act as a test that always passes or always fails respectively.



A special block is the wait until or timeout block. This block will wait until the condition evaluates to true or the specified number of seconds has passed. If the condition evaluates to true within the time limit, the first slot is used, which will often contain a  block. Otherwise, the second slot is used, which will often contain a  block. The two slots can also contain other blocks.

User interaction blocks



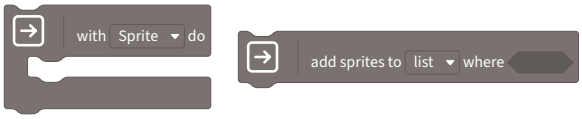
The user interaction blocks will simulate user interaction, similar to the scheduler in Itch (section 5.3.3). Currently, there are blocks to click the green flag, click a sprite, move the mouse, press a key, and answer a

question. The block to click a sprite and the block to press a key also have a synchronous variant, which will wait until all activated scripts have finished, again similar to the synchronous/asynchronous actions in the scheduler from Itch.

Observation blocks

The observation blocks are similar to sensing blocks from Scratch, as they allow the tests to save and query the environment. A special sensing block (which also acts as a variable) reflects the current state of the virtual machine. Test code can then save this state into a variable, making it available later. Two more reporter blocks (one for sprite-specific properties and one for the stage) allow querying specific properties (of specific sprites) from the saved state (or the current state).

Executing blocks in another sprite



While the test blocks can be placed in any sprite, it is our experience that the most convenient place is a dedicated sprite with only test code or in the stage if it does not have code itself. However, this introduces an additional complication: how can the test sprite execute blocks in other sprites? For example, the test sprite might want to move a certain sprite, which is not possible in vanilla Scratch.

To this end, Poke provides a C block, whose contents will be executed in the selected sprite. Technically, this re-uses the broadcasting mechanism built into Scratch. Behind the scenes, the following process happens:

1. Before executing a test, Poke will generate a unique broadcast for each of these C blocks.
2. The contents of the C block are copied to the sprite that will execute them.
3. These copied blocks are placed under a hat block that will trigger when the generated broadcast is sent.
4. The original blocks are replaced with a send broadcast block.

When the virtual machine executes this code, it will send the broadcast, which will trigger the blocks in the correct sprite. This happens transparently for the user: the copied blocks are not visible to the user, and special care has been taken to ensure that there are no performance issues. For example, the blocks are only copied once (or when they are changed).

5.7.3. Feedback in the Scratch environment

For this prototype, showing feedback to students was not a priority. As such, the feedback is shown in a rudimentary interface, which shows the feedback in a tree format (figure 5.5). While usable, it is not the most intuitive layout, especially considering the target demographic of Scratch users.

5.7.4. Comparing Poke to Itch

To verify that Poke is usable as a testing framework, we compared it to the set of Scratch problems from the Flemish Programming Contest (section 5.6.1) that were testable by Itch.

Three of the exercises could not be tested with Poke, but this was expected. Poke does not support extensions at the moment, and these exercises use the *Pen* extension. One of these three exercises would not be testable anyway, for the same reason Itch could not test it: drawing a house is an open-ended exercise. The other exercises could be tested with Poke if the *Pen* extension were supported.

Listing 5.9 shows the Poke test suite for the *Grow and shrink* exercise. For comparison, the Itch test suite for the same exercise is listing 5.1.

5.7.5. Conclusion and future work

Poke shows that it is possible to create a testing framework for Scratch implemented in Scratch. While not as powerful as a JavaScript-based framework like Itch, it is much easier to use, especially for educators without experience in JavaScript. In addition, besides some known limitations, Poke is able to test real-world exercises.

Poke is a prototype, which means that we intentionally did not consider some important aspects, especially for use in educational practice. We identify three major areas where further work is needed.

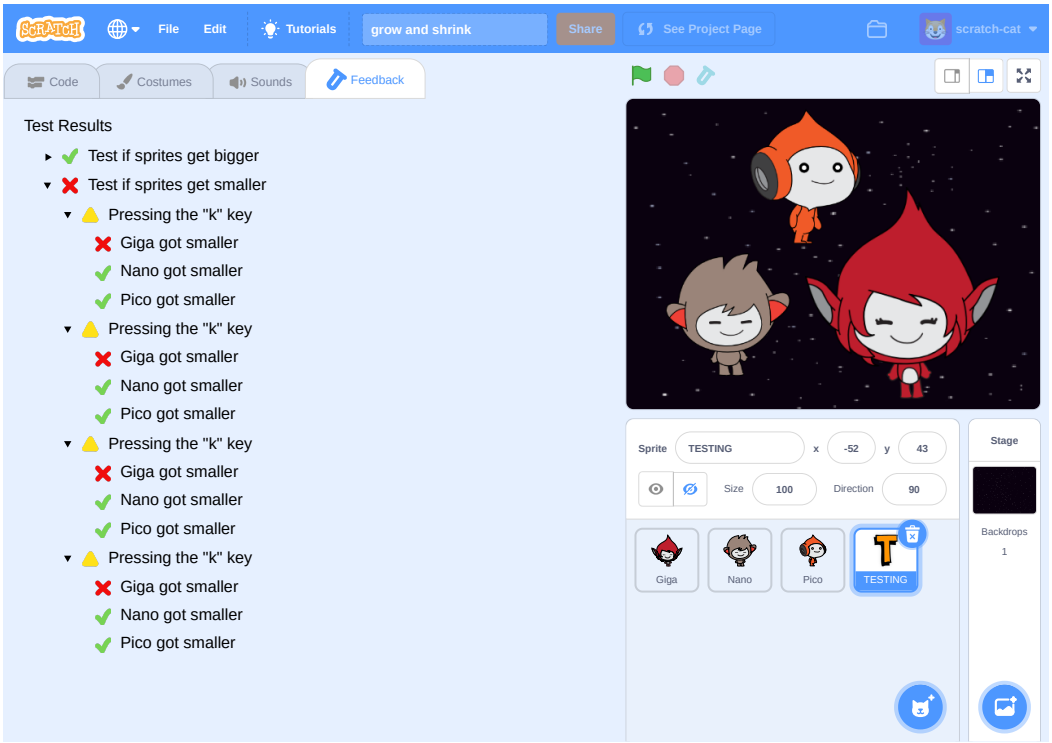
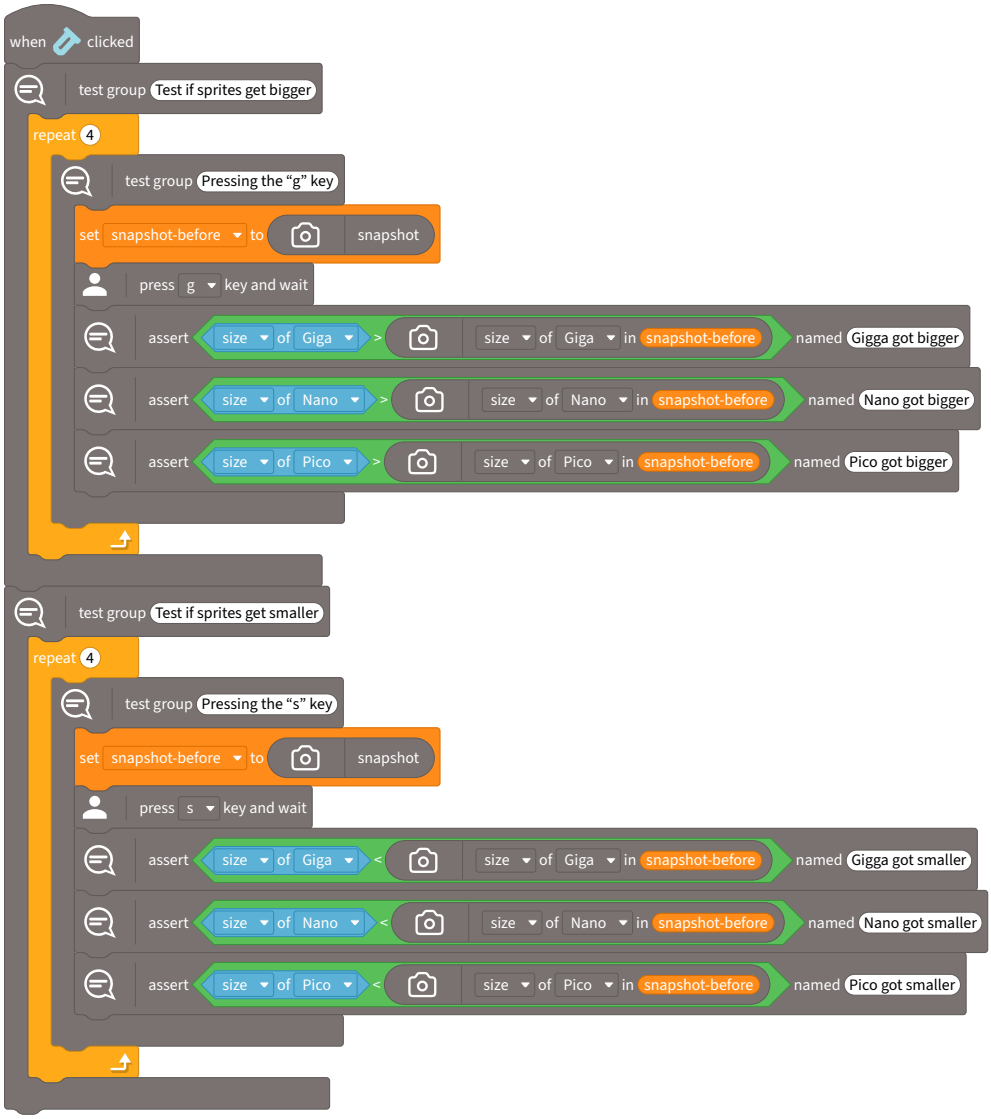


Figure 5.5. Feedback for the *Grow and shrink* exercise. The submission's implementation contains an error: the sprite `Giga` does not implement the shrinking behaviour (with the `s` key). In the feedback, there are two test groups: one for checking that sprites grow and one for checking that sprites shrink. In each of these groups, the relevant key is pressed four times and each sprite is checked. In the "grow" group, everything is correct (indicated by the green tick marks). As expected, the second group reports that some tests have failed (indicated by the yellow triangles). Two of the subgroups are open: while the tests for `Nano` and `Pico` were correct, the one for `Giga` failed as expected (indicated by red crosses). Listing 5.9 shows the test suite that generated this feedback.



Listing 5.9. The complete test suite for the *Grow and shrink* exercise in Poke. The grey blocks are part of the Poke extension. The rendered feedback is shown in figure 5.5.

Support for Scratch features

While normal for a prototype, Poke misses support for various Scratch features. For example, there is no support for sound blocks. The *Pen* extension is the most-used extension and not supported at the moment.

Representation of the feedback

The manner in which feedback is shown to students must be improved. Currently, the feedback is rather text-heavy and dry, which does not integrate well with the game-like nature of Scratch. In addition, we envision that a different representation might be needed for educators and for students.

Organizational aspects of Poke

We have currently not considered any of the organizational aspects that arise when attempting to use a testing framework in an educational setting. The tests are currently included in the Scratch project itself. While useful if students want to create and use their own tests, it is less than ideal for educator-provided test suites. For example, there is currently no way of preventing students from modifying the test code. There is also no support for updating the tests after the fact. For example, in many settings, students receive a "starter" Scratch project in which some code is already present. If the test suite is included in that project, there is no way of updating the test suite after the starter project has been distributed to students.

Poke also has no support for running tests automatically. This would be required, for example, if the Scratch exercises are used as part of an online judge platform.

5.8. Conclusions

In this chapter, we have presented Itch (and Poke) as an educational testing framework for Scratch. With the three phases of the test suites, Itch is able to perform static testing, emulate user interaction, and perform post-mortem testing. Itch provides various helper functions to make testing common scenarios easier. The combination of the three phases

allows for a lot of flexibility in how an exercise can be tested: from fully static to completely dynamic.

We also discussed our experiences using Itch in educational practice. While most exercises can be tested, it remains a challenge to design Scratch exercises that are both easy to test dynamically and open-ended enough to go well with the game-like and tinkering nature of Scratch. It is tempting to use static tests, as they are faster and easier to write. However, while there are good reasons for using static testing, we want to emphasize that dynamic, behavioural testing provides a better experience for students.

Finally, writing test suites in JavaScript is a barrier for educators whose experience is limited to Scratch. We therefore explored a prototype of a testing framework that allows writing test suites in Scratch itself. While writing the tests is technically feasible, some challenges remain. The main one is the organizational aspects of managing these Scratch tests suites in an educational context.

Chapter 6.

A debugger for Scratch

> *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*
> — Kernighan & Plauger, *The Elements of Programming Style*

The process of teaching young students to code is often slowed down by the delay in providing feedback on each student's code. Especially in larger classrooms, teachers often lack the time to give individual feedback to each student. That is why it is important to equip students with tools that can provide immediate feedback and thus enhance their independent learning skills. This chapter presents Blink, a debugging tool specifically designed for Scratch, the most commonly taught programming language for young students. Blink comes with basic debugging features such as 'step' and 'pause', allowing precise monitoring of the execution of Scratch programs. It also provides users with more advanced debugging options, such as back-in-time debugging and programmable pause. A group of students attending an extracurricular coding class have been testing the usefulness of Blink. Feedback from these young users indicates that Blink helps them pinpoint programming errors more accurately, and they have expressed an overall positive view of the tool.

6.1. Motivation and significance

As society becomes increasingly digital, the demand for computer science education is also growing. Current projections suggest that up to 90 % of the workforce will need some level of digital skills to fulfil their professional responsibilities (Bejaković and Mrnjavac 2020). A notable trend in European computer science education is the increasing integration of coding in the curriculum of both primary and secondary schools (Balanskat and Engelhardt 2015). Countries such as Estonia,

France, Spain, Slovakia, and the United Kingdom are leading the way in this integration.

With the increasing emphasis on computer science education, the need for effective tools to facilitate coding education has also grown. The Lifelong Kindergarten group at MIT has played a crucial role in this area by developing Scratch (Resnick, Maloney et al. 2009), an introduction of which is given in chapter 4. Many computer science curricula have adopted Scratch to teach students the basics of computer science, as it is a useful tool for teaching computational thinking concepts (Zhang and Nouri 2019).

As students navigate the programming landscape, they will inevitably encounter bugs that cause unexpected behaviour during code execution (Zeller 2009), and Scratch is no exception. The difficulty in detecting the root cause of a failure is that when a bug is activated, the system will arrive in an erroneous state but might still behave as expected. It is only later when this erroneous state is propagated throughout the system that the system will reach a state that is observably wrong, leading to a failure (Ammann and Offutt 2016). Additionally, Scratch uses a "failsoft" mode: errors are often swallowed, and execution continues without notifying the user (Hromkovič and Staub 2021). Starting from the failure and working backwards to find the bug can be a daunting task, especially in larger or more complex programs.

In a classroom setting, the onus often falls on the teacher to guide students in identifying these bugs. However, the inherent flexibility of programming – where different code can produce the same result – makes this a labour-intensive task for teachers (C. Kim et al. 2018). A tool that makes it easier for students to identify errors is therefore essential, encouraging independent learning and reducing the workload of teachers.

In response to this need, we present Blink, a debugger tailored for Scratch. Blink provides features that allow precise tracking of the execution of a Scratch program, making it easier to identify the cause of unwanted behaviour. By integrating these features into the Scratch programming language, the Blink debugger has the potential to help millions of students learn to code.

6.2. Software description

The Blink debugger is built on top of and adds extra functionality to the Scratch environment (figure 6.1). When the debugger is disabled, all

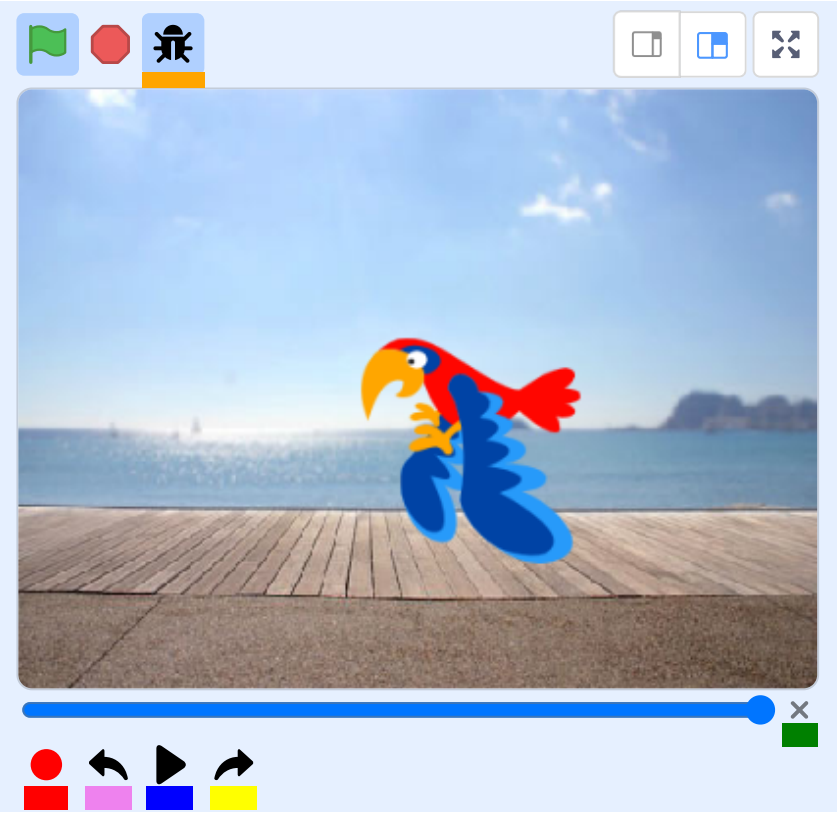


Figure 6.1. Blink additions to the controls and canvas of the Scratch environment. The bug-shaped button (underlined in orange) enables and disables the debugger. It is the only Blink addition that is visible in the environment in normal mode. Other additions become visible when in debug mode. The back-in-time slider underneath the canvas shows the progress of the recording and allows users to jump back and replay the recording. The small button to the right of the slider (underlined in green) allows the user to clear the recording. The red dot (underlined in red) indicates if recording is active by blinking. The back, play/pause, and forward buttons (underlined in pink, blue and yellow) allow the user to control either the recording or the execution.

functionalities of the base Scratch environment remain unchanged. To activate (and deactivate) debug mode, the user must press the bug-shaped button above the canvas. While the debugger is active, the navigation bar turns green as a visual aid. This way, a clear distinction is made between the default Scratch execution mode and debug mode.

6.2.1. Stepwise execution

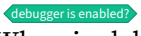
While the debugger is active, the execution of the Scratch project can be paused. In this state, execution can be continued in a stepwise manner, with the step button. Pressing this button will execute exactly one block in all scripts for which there is a corresponding thread in the `Runtime` (see section 6.3.1). After each step, the execution will pause again. This way, the programmer can execute the program at their own pace, providing greater understanding of the impact of each block on the program state and making it easier to identify where errors may occur. To return to the normal execution of the program, the resume button can be used (the same button as the pause button: which changes shape based on the execution state).

To be able to closely follow how a Scratch program is executed, Blink highlights blocks in the workspace with a dark grey hue. During the execution of the program, these are the next blocks that will be executed in each active script. These blocks are also highlighted when replaying a recording of the program.

6.2.2. Back-in-time debugging

Finally, Blink allows replaying the last execution (i.e. the recording) of a Scratch program, inspired by back-in-time debuggers. These debuggers allow the programmer to go back in the execution, often by recording program execution (Balzer 1969; Barr and Marron 2014; Barr, Marron et al. 2016; Chen et al. 2001; Crescenzi et al. 2000; Czaplicki and Chong 2013; Ungar et al. 1997). When the debugger is active, a recording is automatically made, as can be seen on the slider below the canvas. Selecting a point on the slider or pressing the back button will pause execution, and the selected point in the recording will be restored. This includes the entire visual state of all sprites and their clones. While back-in-time debugging is a quite complex debugger feature, our experimental study in section 6.5.2 shows that providing a simple video-player-like interface is intuitive for young students.

6.2.3. Programmed breakpoints

Blink adds three custom blocks to Scratch to manually program breakpoints. Additionally, 

reports whether the debugger is currently active or not. When in debug mode, the execution of the pause block suspends the Scratch program. In contrast, the pause block represents a conditional breakpoint that only pauses the execution if its corresponding condition evaluates to true. The wait until and pause block waits until a condition holds and then suspends the execution.

6.3. Software architecture

Internally, Scratch consists of several interconnected components implemented in JavaScript (section 4.2). However, for the integration of Blink, changes were only made in the virtual machine and the user interface. The interaction between these two components is shown in figure 6.2.

The Blink debugger consists of three main components: instrumentation of the virtual machine for stepping, recording of the execution state, and the custom debug extension, which implements the breakpoints.

6.3.1. Instrumentation for stepping

The virtual machine executes Scratch programs and preserves their current state. It consists of three main modules: the `Runtime`, the `Sequencer`, and the `Execute` module. The `Runtime` module maintains a list of active threads. These objects embody the execution state of a script. Threads are created when certain events trigger (i.e. a green flag is clicked), but can also be removed from the runtime (i.e. when deleting a clone). A Scratch program is executed by repeatedly calling the `_step` method of the `Runtime`, typically 30 times per second. This method subsequently manages threads and calls `stepThreads` as shown in figure 6.2. This will then call `stepThread` for each thread in the virtual machine. This last method will finally call the `execute` method on the `Execute` class, which will do the actual executing.

Blink implements the pause, step, and resume commands of the debugger by instrumenting the `_step` method of the `Runtime`. The instrumentation consists of injecting the code for recording state into this `_step` method in the Blink fork of the Scratch virtual machine. If the

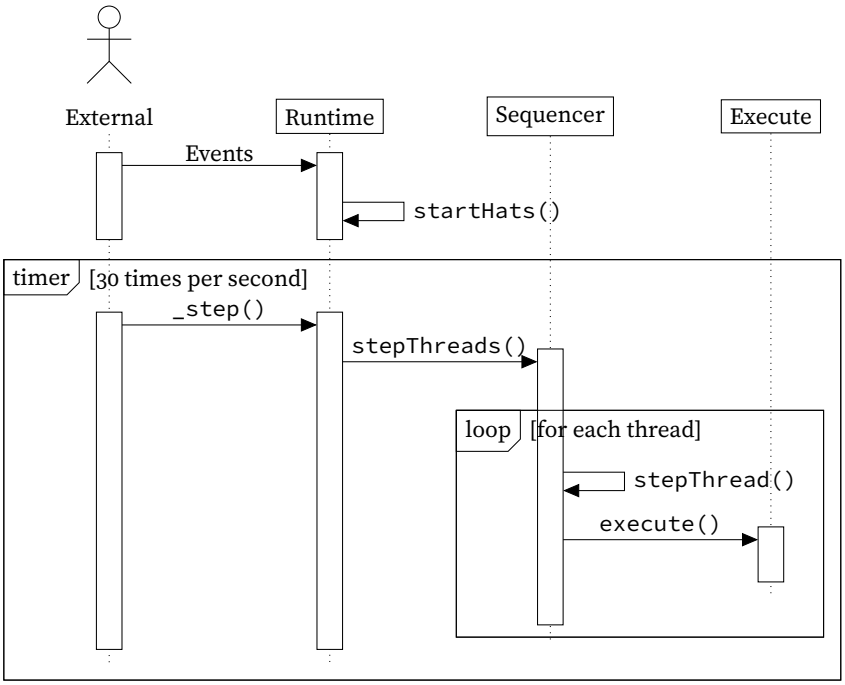


Figure 6.2. Sequence diagram showing the interactions in the virtual machine relevant to the debugger. The `Runtime` receives events from the user interface or from user interaction (grouped as "External" in the diagram), in response to which threads are created or removed. Meanwhile, the `_step` method is called 30 times per second (this is implemented inside the `Runtime`, but should be considered external). This method manages the threads, and calls `stepThreads`, which will then call `stepThread` for each thread in the virtual machine. This last method will finally call the `execute` method on the `Execute` class, which will do the actual executing.

execution of the program is paused, no steps will be taken. It is important to note that all other tasks performed by the `_step` method will still be carried out. Therefore, user actions will continue to result in the creation of threads, which will, however, not be executed.

If, on the other hand, the execution is paused, but a step is taken, the `_step` method will execute one block for each active thread. If the execution is not paused, the `_step` method behaves as normal.

The definition of a step in Blink differs from the `_step` method in the virtual machine and does not execute a single block, as the step functionality does in a traditional debugger. This is a consequence of the Scratch execution model, which presents a dichotomy between code execution and user observable state. The former is inherently sequential: the virtual machine executes one block after the other, decides when a thread switch is needed, and in what order threads are executed. The latter is inherently concurrent: multiple sprites/clones act at the same time.

In Blink, we prioritize maintaining the observed concurrency by ensuring that when stepping through the code, all scripts advance simultaneously for focused debugging. This approach allows users to keep focus on relevant scripts without distractions from visible thread switches or manual stepping, which can be cumbersome in complex programs. Users can thus focus on the script(s) they believe are involved in the failure while ignoring (correct) scripts running at the same time. We discuss this more in great depth in chapter 7.

6.3.2. Back-in-time debugging

Blink also offers back-in-time debugging capabilities. During a debugging session, it instruments the virtual machine and constructs a snapshot of the program state each time a step is executed. These snapshots contain the entire state of the Scratch program, including the position of the sprites, the active blocks, clones, and the pen. To implement this feature, we modified the `Execute` component (figure 6.2) to create a snapshot at the end of the `execute` method. Once a debugging session has been completed, these snapshots can be consulted to restore the program state.

6.3.3. Programmed breakpoints

Pausing the execution of a running program by manually stepping and pausing the execution can rapidly become tiresome. Therefore, many debuggers include the ability to set user-defined breakpoints. In Blink, we have extended the block language to include four additional blocks to pause the execution of a program, to inspect whether the debugger is active and to stop the program based on a condition.

In our original design, we experimented with a large set of highly specific debugging blocks, for example, a block to stop execution when a sprite hits the wall. However, we came to the conclusion that it would be impossible to account for every possible use-case. Therefore, we decided it would be better to provide a small set of fundamental debugging blocks, which can be composed with existing Scratch blocks to express more specific debugging blocks.

6.4. Examples

In this section, we give two illustrative examples to demonstrate how Blink assists in detecting bugs in Scratch programs. The first project consists of guiding a sprite through a maze without going through the walls. The second project is a typical logo exercise where a sprite needs to walk over a star figure without falling into the surrounding water. These projects are typical Scratch exercises where we believe the debugger can be helpful in finding bugs easier and faster.

6.4.1. *Maze* exercise

The *Maze* exercise involves a single sprite in the shape of a red triangle representing the player.[1] The goal of the exercise is to write a program that navigates the player through the maze when the user presses the green flag. Correctly navigating consists of not letting the player navigate through the walls of the maze. In figure 6.3 we show code, which seems to navigate the player through the maze correctly when executing the project. Unfortunately, there is a small mistake towards the end of the solution, which is difficult to see when executing the program at full speed. By making use of the debugger, it becomes easy to navigate through the execution of the program step-by-step, which makes it apparent where the player has taken a misstep. Additionally, due

[1] `https://scratch.ugent.be/blink/editor?project=/blink/maze.sb3`

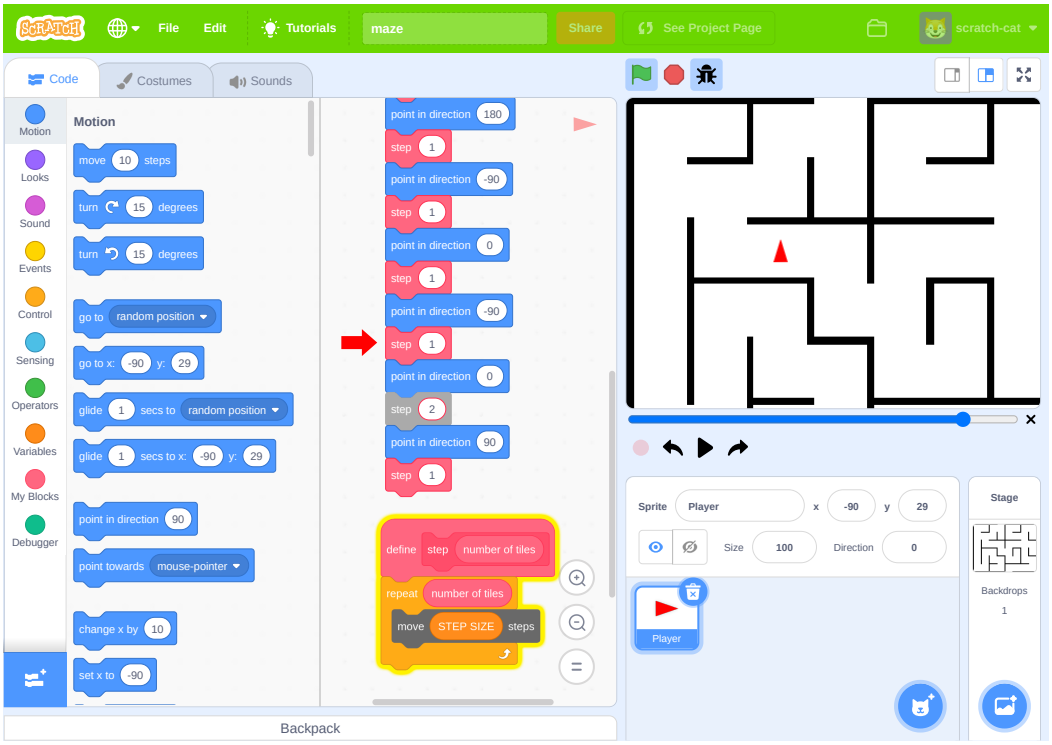


Figure 6.3. The *Maze* exercise. The execution is paused before the failure occurs. As indicated by the light grey colour, we are in the step 2 block. The next block is the move STEP SIZE steps block, as indicated in dark grey. However, since the sprite Player is pointing upwards, it will go through the wall. The bug is in the step 1 block indicated by the red arrow. The solution is to replace this block with a step 2 block: in which case the Player would have moved one square more to the left, meaning the Player can go up without going through a wall. Note that further changes are needed to reach the exit of the maze.

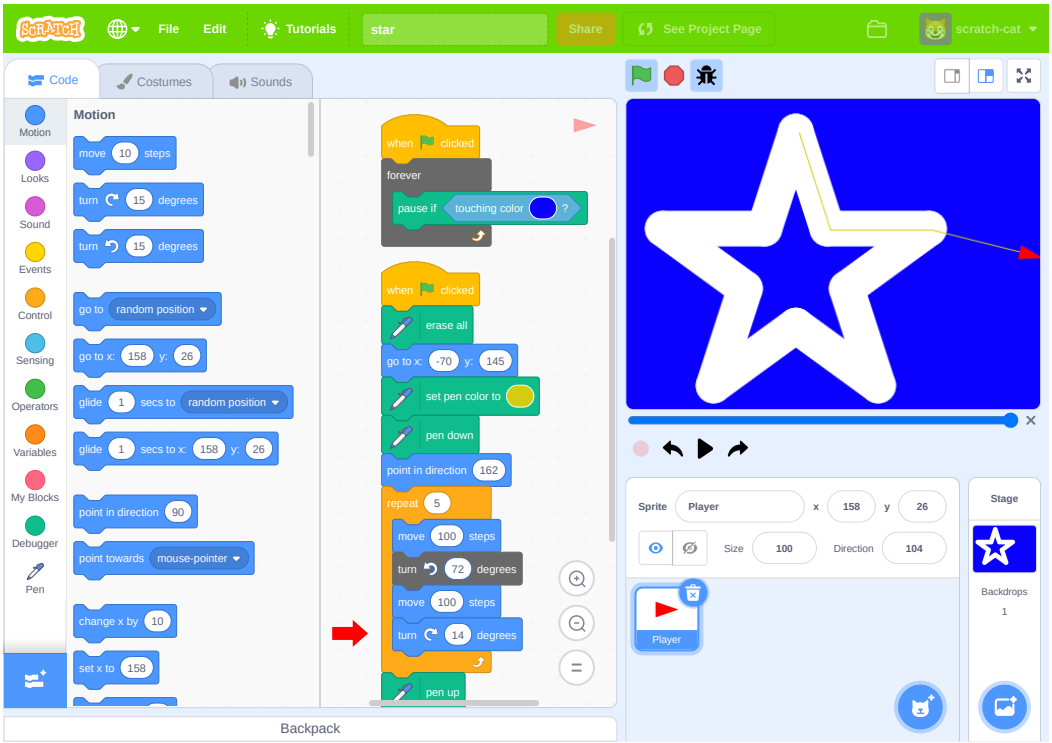


Figure 6.4. The *Star* exercise, which has two scripts. The bottom script contains a bug: the turn 14 degrees block (red arrow) should be a turn 144 degrees block. The top script helps to find the bug: once running, the code loops forever, evaluating the breakpoint each time, and pausing execution if the condition is true.

to back-in-time debugging, it is easy to trace back to precisely where the bug emerged in the code. With more traditional debuggers, which only provide step-wise execution, programmers frequently discover program errors too late, necessitating restarting the complete debugging session.

6.4.2. *Star* exercise

While the *Maze* exercise demonstrates the capabilities of Blink in regard to step-wise execution and back-in-time debugging, the *Star* exercise demonstrates the helpfulness of our custom debugging blocks.[2] The aim of this exercise is to walk over a star-shaped figure surrounded by

[2] `https://scratch.ugent.be/blink/editor?project=/blink/star.sb3`

water. Any program that directs the player into the water is buggy. By making use of Blink's custom debugging blocks, it is quite easy to pause the execution of the program when the player touches something blue (figure 6.4). As soon as the green flag is clicked, the code repeatedly checks whether the player is touching something blue. If it does, program execution is paused by using the `pause if` block.

6.5. Impact

6.5.1. Related work

While most textual programming languages have debuggers, the emphasis in this chapter is on block-based languages. In this domain (excluding Scratch), the most notable debuggers are the Microsoft MakeCode Arcade (Ball et al. 2019) debugger and the Blockly debugger (Savidis and Savaki 2020). Both of these debuggers offer basic debugging facilities, such as breakpoints, step functionality, and variable watches. Unfortunately, these debuggers are not applicable to the Scratch environment because both MakeCode Arcade and Blockly assume a sequential execution model, while the Scratch programming language is inherently concurrent. The Snap*!* block-based programming language (Mönig and Harvey 2024) does support concurrency, but its debugger lacks back-in-time debugging functionality. Next to block-based programming languages, we have taken inspiration from back-in-time debuggers (Barr and Marron 2014; Barr, Marron et al. 2016).

For Scratch, we are aware of three existing debuggers. The first provides pause/resume/step functionality and breakpoints (B. L. Wang and Klopfer 2021). The second debugger is a browser extension that provides breakpoints and advanced logging (*Scratch Addons* 2023). The last, and most recent, debugger is NuzzleBug (Deiner and Fraser 2024). It provides similar functionality to Blink, but makes some different choices in its implementation and user interface. For example, the step functionality in NuzzleBug is a more traditional step, advancing one block at a time.

The need for tools to help find bugs in Scratch projects is also illustrated by the existence of other tools. These include multiple linter-style tools like Hairball (Boe et al. 2013), Dr. Scratch (Moreno-León and Robles 2015), QualityHound (Techapalokul and Tilevich 2017) and LitterBox (Fraser et al. 2021), or test frameworks such as Whisker (Stahlbauer, Kreis et al. 2019) and ITCH (Johnson 2016).

6.5.2. Experimental study

To validate the usefulness of the debugger in helping students understand their code better, we conducted a quasi-experimental study (Shadish et al. 2002). The test group consisted of 16 students aged 8 to 11, which is at the lower end of Scratch's target audience.

Our experiment started with a short introduction explaining the different features of Blink and how they can be used. Subsequently, we provided the students with two projects as discussed in section 6.4. Both projects contained an error leading to unwanted behaviour during the execution of the program. During each exercise, we encouraged the students to make use of the features of the debugger to correct the implementation.

Our study aimed to determine the ease of use and usefulness of Blink. Participants were asked to rate both aspects for three different aspects of the debugger tool: traditional debugger operations, usage of breakpoints, and back-in-time debugger. A five-point scale, represented as a row of smiley faces, was provided for each statement. The results of this questionnaire are shown in figure 6.5.

The findings indicate unanimous agreement among participants that revisiting previous states is a straightforward and advantageous approach for identifying errors in Scratch code. The custom breakpoints were less intuitive for most students and received a lower score on ease of use. While breakpoints were conceptually clear to most students, when they were asked to implement a custom breakpoint to pause the execution, many of them struggled. Finally, if we look at the score distributions for the debugger in general, we can see that they embody the combination of the distributions for the usage of the breakpoints and the ability to go back in time.

6.6. Conclusions

In this chapter we introduced Blink, a debugger for the Scratch programming language. Blink offers stepwise execution, back-in-time debugging, and the ability to define custom breakpoints. These features allow students to closely follow the execution of their programs and consequently make it easier to find errors. To evaluate the effectiveness of our debugger, we conducted a quasi-experimental study. The results of this study show that most aspects of the debugger are both useful and easy to use. We observed a strong preference for back-in-time debugging and observed

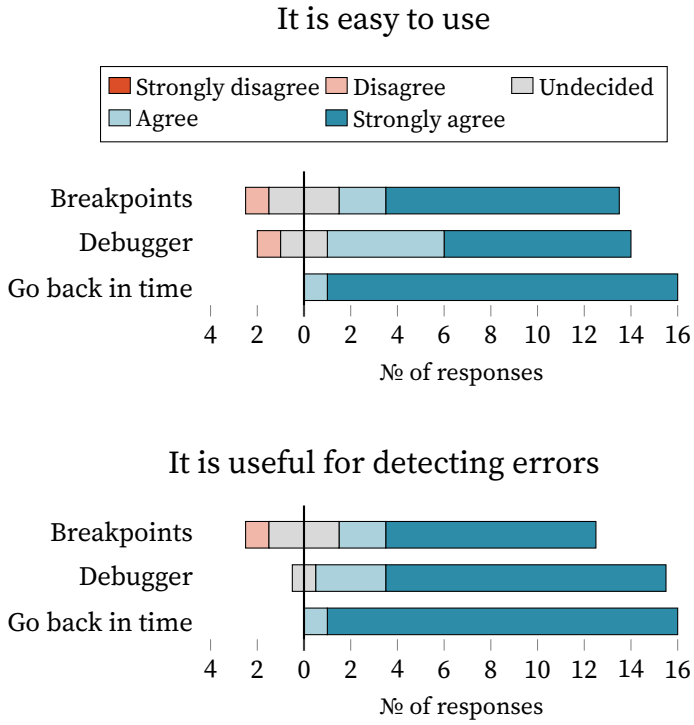


Figure 6.5. Answers regarding the ease of use and the usefulness of the debugger in general, the breakpoints and the ability to go back in time to detect errors.

that custom breakpoints are less intuitive than back-in-time debugging. In future work, we will focus on inventing novel user interface elements to further improve the ease of use for programming custom breakpoints for students.

Chapter 7.

The Scratch execution model

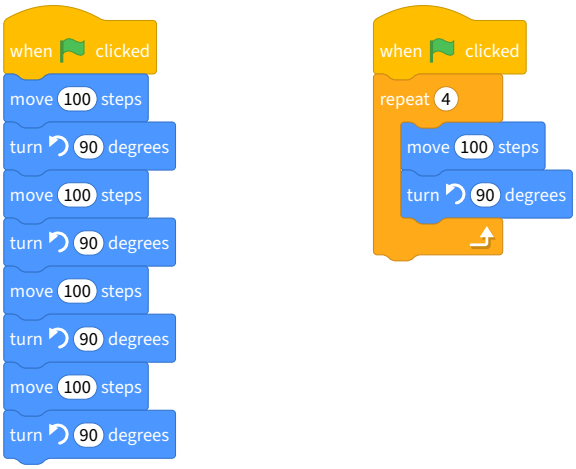
> *Concurrency is often considered an advanced programming technique. Yet our everyday world is highly concurrent, so Scratch users are not surprised that a sprite can do several things at once.*
> — Maloney & Resnick, *The Scratch Programming Language and Environment*

The programming language Scratch inherently supports parallel programming. Each Scratch program (called a project) consists of a number of sprites with individual code (chapter 4). Blocks that are attached together form scripts, and each sprite can have multiple scripts running concurrently as separate threads within the Scratch virtual machine.

The Scratch execution model combines a fixed-step time loop (30 frames per second) with an almost-cooperative threading model. This means that threads are seldom interrupted, mostly relying on explicit yielding to other threads. While this approach minimizes the occurrence of certain race conditions, some concurrency issues persist (Maloney et al. 2010). For instance, the order in which sprites respond to broadcasts can be unpredictable. Consequently, even without explicit concurrency controls, Scratch is a useful tool for teaching concurrency concepts (Fatourou et al. 2018).

However, the current execution model has some drawbacks. The cooperative nature of the threading model can lead to unexpected behaviour when working with multiple sprites. Additionally, the execution model complicates the use of debuggers within Scratch. A traditional step function (which steps one block at a time) exposes execution states that are normally hidden. Alternative step functions (see, for example, the one used by Blink in section 6.2.1) diverge from the normal execution and are thus undesirable.

To address these issues, this chapter begins with an in-depth exploration of Scratch's current execution model. This analysis is essential for understanding the subsequent section, where we explore the model's



Listing 7.1. Two Scratch programs that seemingly exhibit the same behaviour: the sprite moves in a square of 100 steps, and finally stops at the same position as the start of the program.

shortcomings in more detail. We then propose some modifications to the execution model, which would solve the issues we have identified. These modifications are finally evaluated to determine their impact on real-world Scratch projects in a preliminary benchmark.

7.1. Elements of a Scratch program

A Scratch program consists of zero or more sprites and a stage (see also chapter 4). For every sprite at least one target is created (a target is what is drawn on the screen), while the stage has exactly one target. All targets have their own local state: the variables and visual properties (e.g. position, size, bounding box, colour, direction). Clones create more targets of the same sprite. While clones have their own separate state, all targets based on the same sprite share the same code. In the virtual machine, there is no substantial difference between how targets from different origins (sprites, stage, clones) are handled, so we can just consider targets for the remainder of this chapter.

Code-wise, the Scratch blocks are organized into categories (see section 4.1.1). However, in this case, it is useful to look at their technical type, which corresponds to their shape. In total, there are seven types of blocks:

1. Hat blocks  , which are placed at the start of a script (they are named hat blocks since they visually sit on top of a script). A script can only have one hat block. They function as event listeners, which trigger execution of the script if the event occurs.
2. Stack blocks  , representing program statements. These are the most common blocks. They are called stack blocks since they are stacked on top of each other. Stack blocks broadly fulfil the role of statements in Scratch.
3. C blocks  , which are named after their shape. They are used for most of the control flow blocks: loops and branches. The variant for the if/else block is sometimes called an E block since it has two slots.
4. Reporter blocks  , act as variables or values and can be slotted into other blocks. Operators that result in a value also have this shape. The reporter blocks fulfil the role of expressions.
5. Boolean blocks  , which are analogous to reporter blocks but result in a boolean.
6. Cap blocks  , which end a script: no blocks can be added afterwards. Note that the infinite loop block, for example, is both a C block and a cap block.
7. Custom blocks 

, which define “procedures”.

7.2. Related work

The Scratch execution model is defined by its implementation in the virtual machine. There exists, at least to the knowledge of the authors, no comprehensive formal description of the execution model. This does not mean there is no prior work. From the Scratch team, Maloney et al. (2010) provide a high-level description of the threading model.

Another body of works that provides insights into the Scratch execution model comes from the *Chair of Software Engineering II* group, led by Gordon Fraser. These publications all provide descriptions for parts of the execution model.

First, Stahlbauer, Kreis et al. (2019) propose a formalization of three aspects in Scratch: the user perspective, a syntactic model and a semantic model. They describe the semantics of Scratch with a memory model based on message passing. Next, Stahlbauer, Frädrich et al. (2020) develop LeILa, an intermediate language to which Scratch projects can be translated, with the intended use of performing analysis on Scratch

projects. The authors also provide a formalization of LeILa, using approximations for the behaviour of Scratch in some areas. Also, Gotz et al. (2022) model the state-based behaviour of Scratch programs using a finite state machine. Finally, Deiner, Feldmeier et al. (2023) delve deeper into the actual execution of the virtual machine, while also proposing some modifications to it, for example, to make execution deterministic.

Other block-based languages also have to deal with concurrency. For example, MakeCode also uses a non-preemptive threading model, inspired by Scratch (Ball et al. 2019). There has also been some work on concurrency and concurrency controls in other block-based languages (Chung et al. 2020). However, since these languages do not use the Scratch virtual machine for execution, their relevancy for this chapter is limited.

7.3. The current execution model

7.3.1. Execution of a Scratch program

Arguably, it is more of a concrete syntax tree, as e.g. the position of blocks is also saved. However, in Scratch's case, the differences are minimal, so we call it an abstract syntax tree, as Scratch themselves do.

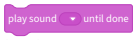
When executing Scratch code, the virtual machine transforms the blocks into an abstract syntax tree. These are organized by target, and every execution of a script results in a distinct thread inside the virtual machine. These are green threads: implemented fully in the virtual machine.

The virtual machine is thus responsible for scheduling these threads. Figure 6.2 gives a schematic overview of the interaction between the different parts. It uses an almost-cooperative threading model, which Maloney et al. (2010) call the "Scratch threading model". This means it is mostly non-preemptive: the virtual machine will not interrupt threads at arbitrary points in their execution. The threads must voluntarily yield control or reach a limited set of points in their execution. The rational is given in Maloney et al. 2010: "Scratch builds concurrency control into its threading model in a way that avoids most race conditions, so that users do not need to think about these issues. This is done by constraining where thread switches can occur.".

At four well-defined points, a thread always yields, thus causing said thread switching:

1. When a block with a fixed duration is executed. There are a number of blocks that fall under this category. 

is an obvious inclusion, but this also applies to 

, for example. 

also falls under this category, even if there is no explicit time.

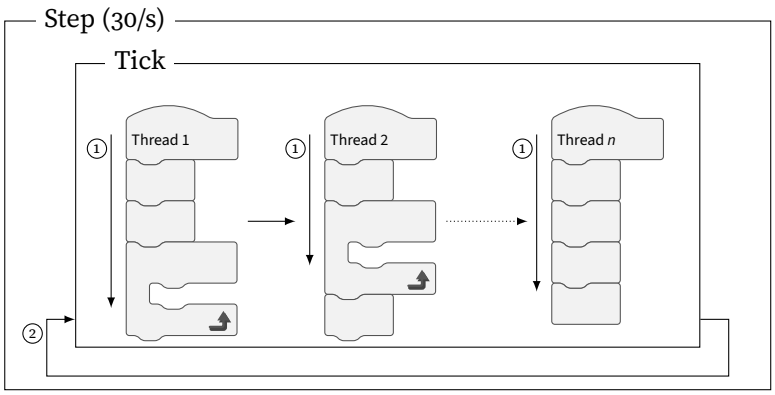
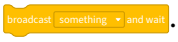


Figure 7.1. Overview of the interplay between the threading model and the "game loop". Within one step (which is done 30 times per second), one or more ticks are executed. The arrow with ② illustrates this: after the first tick, another is started if less than 75 % of the step time (the time one step has to complete, 33 ms) has been used, and a redraw has not been requested, and Scratch is not in turbo mode. Within one tick, a turn is executed for each thread ①: a thread executes until it terminates or the thread yields.

2. When a block waits on execution of other blocks. For example,  .
3. The last block of a loop (thus  ,  , and ). This means thread switching will occur after every loop iteration.
4. A recursive procedure call is detected. Scratch attempts to detect these (up to five levels of indirection) and will yield the thread on each call if it detects a recursive call.

There is one exception: when using procedures "without screen refresh", Scratch will interrupt a thread that runs longer than 500 ms. This is called the "wrap timer", and has some curious edge cases.[1]

The threads are executed in a first-come, first-serve manner: there are no priorities nor changes in thread order. The first thread is executed until it yields or ends, then the next thread, and so on. We call the execution within one thread until it yields or ends a **turn**. A thread can have one of three conceptual states: *done*, *running*, and *yield*.

The virtual machine uses a *fixed-time step with synchronization* main loop (Nystrom 2014), also called a *synchronized coupled model* (Valente

[1] `https://github.com/scratchfoundation/scratch-vm/issues/2834`

Scratch 3 should actually run at 60 fps; however Scratch enables a compatibility mode by default, resulting in 30 fps.

et al. 2005). This means that the virtual machine runs in **steps**: internally, the `step` function is called every 33 ms (so 30 times a second, commonly known as 30 fps).

In each step, the virtual machine will execute one or more ticks. A **tick** is one turn in every thread: the first thread is executed until it yields or terminates, then the second thread and so on. After the tick, a redraw is performed if needed (in practice this is always done, as the source code contains a to-do to implement selective redrawing). After the first tick is finished, the virtual machine decides whether to run another tick (figure 7.1). A new tick is started if less than 75 % of the step time (the time one step has to complete, 33 ms) has been used and a redraw has not been requested. Note that once a tick has started, it is run completely and cannot be stopped. The arbitrary 75 % is intended to prevent frame drops: steps that take longer than their allocated step time, meaning the next step is delayed. Also, in practice, many blocks request a redraw, so in many Scratch projects, a step only ever runs one tick.

A concrete example of the execution model is given by figure 7.2, which shows the execution of the programs from listing 7.1. These programs have only one thread. Since none of the blocks in the unrolled program yield the thread, the full program is executed in one tick. In the other version, with a loop, the thread yields after each iteration of the loop, meaning the program needs four steps. This does result in an observable difference: in the unrolled program, the sprite does not move visually. As a redraw only happens between steps, the sprite is back at its original position. In the looped version, the sprite moves four times (albeit rapidly), as there is a redraw between each step.

7.3.2. Implementation details

How different parts of the virtual machine implement the execution model from section 7.3.1 is shown in figure 6.2. When the user interface loads a project, it also starts the virtual machine. This means that the game loop is active (this is done in the class `Runtime`).

New threads are only created in two scenarios:

- Code needs to be executed, either because an event triggered some hat blocks (green flag, key press, etc.) or because the user clicked on some blocks.
- When “stage monitors” or watchers are active. These are used in the user interface to show the value of variables or properties. The watchers for variables also allow the user to change the value of the variable.

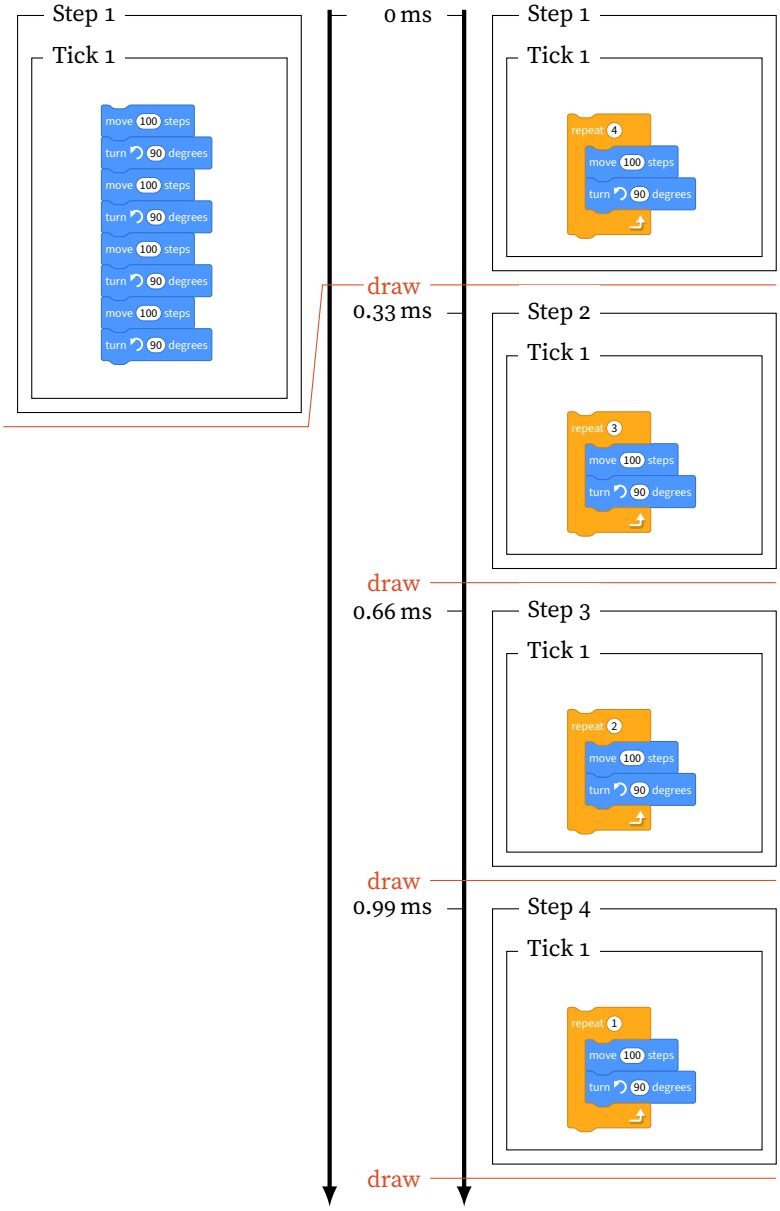


Figure 7.2. The execution of the two programs from listing 7.1. In the unrolled version (left), all code is executed in the first turn, meaning only one tick and step is needed. In the version with loop (right), the loop yields after each iteration, meaning the rest of the step is filled with idle time. In total, four steps are needed.

The `Runtime` calls the method `stepThreads` in the `Sequencer` class. This class is responsible for implementing the ticks. After each tick, done threads are removed, and a new tick is started if possible. It is also here that the thread status is managed. While there are three conceptual statuses, the implementation has five:

`done` The thread has finished executing all blocks and will be removed after this tick.

`running` The thread is being executed and has more blocks to execute. It will be scheduled again next tick.

`yield` The thread is waiting an amount of time. The thread is scheduled again next tick to see if the wait time is over.

`promise wait` The thread is waiting for a JavaScript promise to be resolved, after which the thread will be set to `running`.

`yield tick` The thread yields until the next step. The purpose of this status seems to be some performance optimizations to aid with benchmarking.[2]

Each thread maintains a stack structure. The `Sequencer` will then look up the next block on said stack, and if there is one, it will call the `Execute` class. That class will actually execute the block on the stack. For normal blocks (somewhat confusingly called "stack blocks", since they stack together to form a script), the current block is popped from the stack, the block is executed, and the next block is put on the stack. The stack is only useful when working with C-blocks or procedures. For example, C blocks will push the first block in their slot on the stack. In the case of a loop, a counter is saved in the stack frame to determine how many times the loop should be run. The exact implementation of the stack is less relevant for this chapter, so it is left to the reader to browser the source code.

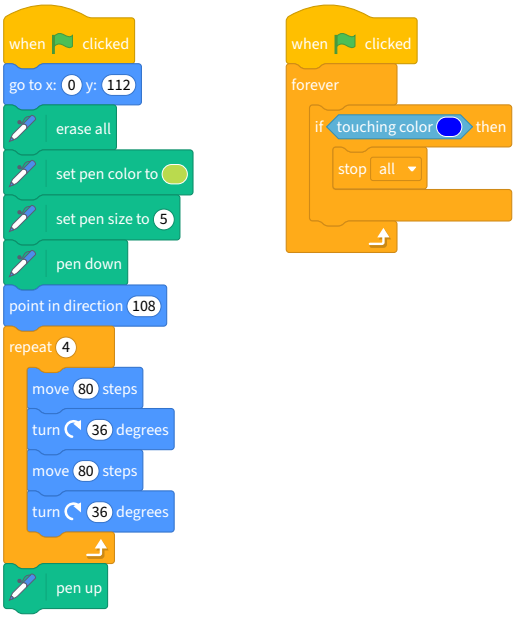
7.4. Limitations of the execution model

This section illustrates a few limitations of the current execution model, first in general and then specifically for a debugger.

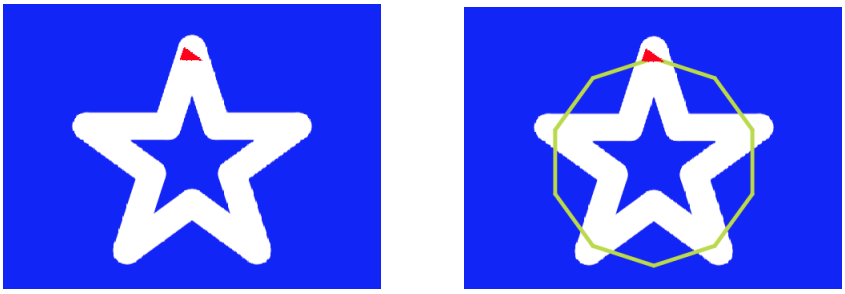
7.4.1. During general execution

As Maloney et al. (2010) mentioned, the Scratch threading model does not solve all issues with concurrency. To illustrate this point, we will

[2] `https://github.com/scratchfoundation/scratch-vm/pull/1211`



Listing 7.2. The implementation, with a bug in the first script (left) and a non-working second script (right).



(a) The stage before execution. **(b)** The stage after execution.

Figure 7.3. Result of running the implementation from listing 7.2 for the *Star* exercise.

consider a variant on the *Star* exercise from section 6.4.2. In this exercise, the goal is to let a sprite move around on a path without falling into the water (read: touching a blue colour). Listing 7.2 shows an implementation for this exercise with an additional script that will stop execution if the sprite touches something blue. We asked a number of educators that had experience with Scratch to predict the behaviour of this implementation. All of them expected the execution to stop either when the sprite first touches the water or after executing the block when the sprite first touches the water. However, as can be seen in figure 7.3, where the canvas is shown before and after running the code, the second script that should have stopped execution did not work.

The reason for this is the non-preemptive thread switching: the body of the loop is always executed atomically. At the start of the loop, the sprite does not touch the water. After execution of one iteration, the sprite is back on the path and does not touch the water. Therefore, whenever the second script is executed, the sprite is not touching the water, which explains why the execution was not stopped.

Consider the original code in the *Star* exercise from section 6.4.2. The second script there does not stop the execution, but uses Blink's pause block to halt the execution. Using the current execution model, the pause block will not function for the same reasons mentioned above.

7.4.2. Specifically for a debugger

A fundamental feature of any debugger is the ability to step through code: executing one statement and then pausing the execution to facilitate inspection of the program state. The functionality is also essential in debuggers for Scratch: all existing debuggers for Scratch implement it. In Scratch, executing a single statement translates to executing a single block.

However, traditional single-block stepping has some drawbacks in Scratch. A first drawback is that users have to click a lot, since the stepping functionality is global, not per thread. Secondly, and more importantly, the step functionality exposes details of the Scratch execution model to the users. For example, thread switching, which is normally implicit in Scratch's perceived parallel execution, becomes visible to the users during debugging.

Additionally, debuggers must choose what to do with intermediate states that are typically hidden during normal execution. For instance, five consecutive blocks would result in one redraw in the normal execution.

One choice is to not alter the redraw logic (thus only redrawing when the normal execution would redraw), but this results in steps having no visual impact, even if the block logically should change the visuals. The other choice is to redraw after every step. While the effect of every block (and step) is then visible, this exposes intermediary steps that would normally not be drawn. Some blocks (for example, checking if a sprite touches a colour) use the visual state, meaning these additional redraws can result in a different execution of the project.

As described in section 6.3, our debugger Blink takes a different approach to the stepping feature. We believe it is useful to maintain the observed parallelism of Scratch in the debugger: we define a step in the debugger as executing one block in every thread. This approach allows users to keep focus on relevant threads without distractions from thread switches, which can be cumbersome in complex programs. Users can thus focus on the script(s) they believe are involved in the failure while ignoring (correct) scripts running at the same time.

This approach does come at a price: it changes the Scratch execution model, which is not trivial due to two main considerations:

1. If the execution model is only changed when debugging, the debugger does not debug the same program execution as when running the program. This can result in different behaviour, meaning the bugs for which the debugger is used need no longer be present or new bugs, unique to the debugger, could be introduced.
2. If the execution model is changed, we need to ensure that existing Scratch programs keep working and that we do not introduce concurrency problems, as the current execution model of Scratch was explicitly chosen to avoid those.

We opt for the second option: modifying the Scratch execution model. In the next section, we discuss what we changed, after which we investigate the impact on performance and behaviour of existing Scratch projects.

7.5. Towards a new execution model

In line with how we want the stepping feature of the debugger to work, we have decided to change the Scratch execution model as follows: we modify a turn to execute exactly one block before yielding. Thus, in a single tick, the virtual machine will execute a single block in every thread.

As there is often only one tick per step (due to many blocks requesting redraws), this means that only one block would be executed per step (thus one block for every thread per 33 ms). Consequently, this makes execution slower than in the original execution model.

A possible remedy is to modify the number of steps that are taken. For example, it might be better to run at twice or more times the number of steps per second. In turbo mode, this would mean the steps are done as fast as the hardware allows. While this does make everything go much faster, it does introduce a big difference in execution time between a performant machine and a slower machine.

These changes to the execution model have as a benefit that the *Star* solution (listing 7.2) will behave as expected, since the first thread will yield after the first block in the loop. It can also illustrate that this does introduce concurrency considerations that were not present in the original execution model. For example, if the conditional block in the second thread evaluates to true, the first thread will execute another block before execution is stopped by the block inside the conditional block from thread two.

In the next section, we analyse existing Scratch projects to determine which frame rate for the new execution model most closely results in the same execution speed as the original execution model (which runs at 30 frames per second). We also take this opportunity to analyse the complexity and block use in Scratch projects, to evaluate whether the concurrency considerations would cause problems.

7.6. Exploration of Scratch projects

As Scratch is used by many people, it is important that changes to the execution model do not adversely affect existing Scratch projects. However, this requires us knowing what Scratch projects look like. The aim of this analysis is to determine what blocks are used in Scratch projects, how big projects are, and what programming concepts are used.

We begin by looking at existing work on analysing Scratch projects, followed by our own analysis.

7.6.1. Existing analyses

Aivaloglou and Hermans (2016) analysed 250 000 Scratch 2.0 projects they scraped from the public Scratch site. They looked at the types of blocks

used, the size of the projects, and the complexity. For the complexity, they utilize the cyclomatic complexity metric (McCabe 1976). The considered decision points are the `if` and `if-else` blocks.

There is some discussion if the cyclomatic complexity is a useful metric. It might have no more predictive ability than lines of code (Cherf 1992; Fenton and Neil 1999; Hatton 2008).

They found that most Scratch projects are small: 75 % have less than 5 sprites, 12 scripts, and 76 blocks. 25 % has less than 12 blocks, although there are some huge projects with more than 20 000 blocks. They also found that about 78 % of projects have no decision points.

Fronza et al. (2020) investigate Scratch projects with different complexity metrics. The dataset is, however, much more limited: 80 projects were analysed. The authors also measure the cyclomatic complexity, in addition to some Halstead complexity measures (Halstead 1977), and their own proposal for a "when" metric. They do use more decision points for the cyclomatic complexity (`if`, `if-else`, `repeat until`, `wait until`, `and`, `or`). The proposed "when" metric counts the number of "when" blocks (e.g. hat blocks with certain conditions).

The usefulness of the Halstead metrics is even more controversial (Hamer and Frewin 1982; Jones 2019; Shen et al. 1983).

7.6.2. A new dataset of Scratch 3.0 projects

Since the dataset used by Aivaloglou and Hermans (2016) consists of Scratch 2.0 projects and Fronza et al. (2020) only analyse 80 projects, we found it necessary to collect a new dataset of Scratch projects.

We constructed a new dataset as follows, using the Scratch website.[3] Creating a new project provides an identifier (996725074, April 7th, 2024), which we used as a starting point. We then subtract one from the identifier, downloaded the project if possible, and continued. The oldest project in the dataset is from April 5th, 2024, with identifier 995595608.

The 1 129 465 projects were created in approximately 3 days, illustrating Scratch's popularity.

This resulted in 237 926 downloaded public projects (of the total 1 129 465 that were made between our newest and oldest projects). From those, 207 were corrupt or for an older version of Scratch. We also filtered out the following projects: 37 936 (15.9 %) were empty and 4411 (1.9 %) had no executable code (e.g. only head blocks or scripts without head blocks). This results in a final dataset of 195 372 Scratch projects we considered for further analysis.

7.6.3. Analysing Scratch 3.0 projects

Hairball (Boe et al. 2013) was commonly used to analyse Scratch projects (it is also used by the existing analyses), but does not support Scratch 3.0.

[3] `https://en.scratch-wiki.info/wiki/Scratch_API`

For this reason, we have implemented a similar tool in JavaScript (versus Hairball's use of Python).[4] It also supports plugins to support extensions for other analyses. Being written in JavaScript, it has the advantage that it can reuse parts of the Scratch virtual machine, like reading and parsing Scratch projects.

Wherever possible, we have used the same definitions and metrics as used by Aivaloglou and Hermans (2016), for ease of comparison. Whenever a direct comparison is possible and relevant, we have included their data in *italics*. For example, 75 % (*60 %*) indicates our data shows 75 %, while Aivaloglou and Hermans found 60 %.

7.6.4. Use of blocks

Scratch blocks can be categorized into seven types, based on their shape (section 7.1), their usage shown in figure 7.4. Blocks can also be put into categories (section 4.1.1). The number of projects that use a block from a certain category is shown in figure 7.5. Note that these numbers do not fully compare with Aivaloglou and Hermans (2016): since then, some new Scratch extensions were added, and the pen-related blocks have moved to an extension.

It is notable that extensions are not widely used: only 12.2 % of projects use any extension. The Pen extension is the most popular one, appearing in 6.9 % of the projects.

7.6.5. Size and complexity

Table 7.1 shows a summary of the project size for our dataset. In the rest of this subsection, we detail some choices we made in analysing the projects and draw some conclusions.

When considering the size of a program, a frequently used metric is lines of code. However, there is no universal agreed-upon manner in which to count lines of code (Nguyen et al. 2007). Two variants are frequently used: physical lines of code (the number of lines in the source files) and logical lines of code (an approximation of the number of statements or expressions). While normally counting the physical lines of code is easy and counting logical lines requires some consideration, the reverse is true in Scratch. To count logical lines of code, we can simply count all blocks. For physical lines of code, we chose to count the number of

[4] `https://github.com/scratch-ed/scratch-analysis`

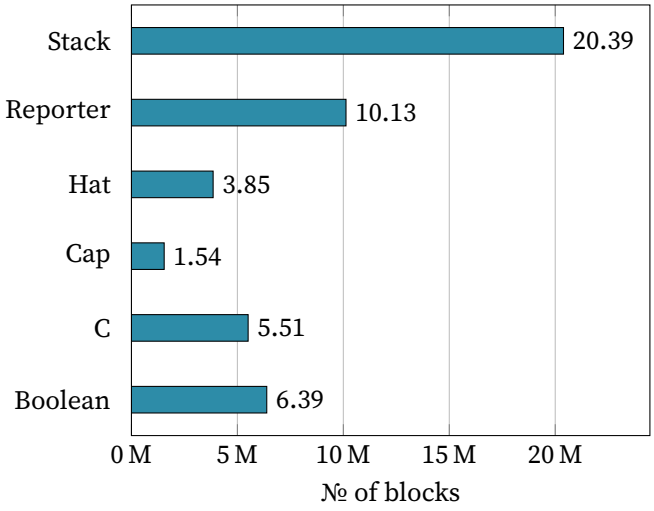


Figure 7.4. Number of blocks by shape in all projects. The total number of blocks is 47 808 628. Note that the *forever* block is counted twice (as a cap block and a C block), and procedure-defining blocks are counted as hat blocks.

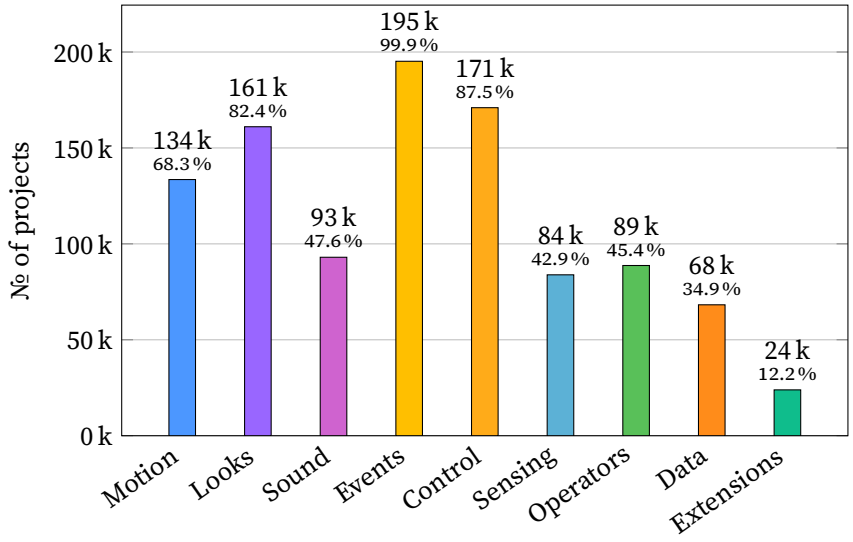


Figure 7.5. Number of projects that use blocks from a certain category. Custom blocks are excluded, and all blocks for extensions are counted together. Bar colours correspond to the block category colours in the traditional Scratch 3.0 colour scheme.

Table 7.1. Size and complexity statistics about the 195 372 non-empty Scratch projects in our dataset. Unless otherwise noted, all numbers are shown per project and blocks are counted as logical lines of code. The first column of numbers reports the mean from Aivaloglou and Hermans (2016) if available. The subsequent numbers are, in order, the mean and the five-number summary: the minimum, the first quartile, the second quartile (the median), the third quartile, and the maximum.

	Aivaloglou et al.	mean	min	Q_1	Q_2	Q_3	max
sprites (with code)	5.68	4.98	1	1	2	5	1000
scripts (with code)	17.35	19.73	1	2	4	11	9134
blocks (logical lines)	154.55	203.14	2	9	22	75	24 084
blocks (physical lines)	N/A	150.81	2	8	20	65	20 249
dead blocks	N/A	42.25	0	0	0	3	14 912
blocks per script	N/A	10.30	1	2	5	10	5497
cyclomatic complexity per script	1.58	1.85	1	1	1	2	5497

"main" blocks in a script. This means we do not count blocks used as arguments, e.g. the condition block of a loop is not counted.

When counting the number of scripts per project, we excluded scripts that consist only of a hat block, as these do nothing. We similarly excluded sprites without code from the count of sprites per project. These can have a role in some cases but are not useful in the statistics.

From these data, we can see that 75 % of projects have less than 5 sprites, 11 scripts, and 80 blocks (*75 % of projects have less than 5 sprites, 12 scripts, and 76 blocks*). We can conclude that most Scratch projects are still small in Scratch 3.0.

For the cyclomatic complexity, we used the same decision points as Aivaloglou and Hermans. Figure 7.6 shows the distribution of the cyclomatic complexity in the Scratch projects. Most scripts (72.6 %, *78 %*) do not contain any decision points. Additionally, another 15.3 % (*13.8 %*) has just one decision point. On the other end of the spectrum, 1.6 % has a complexity larger than 10, and 667 scripts (0.000 17 %, *0.000 052 %*) have a complexity larger than 100.

We can conclude that most projects are small, since 75 % has less than 5 sprites, 11 scripts, and 80 blocks. Most scripts (72.6 %) have no decision points.

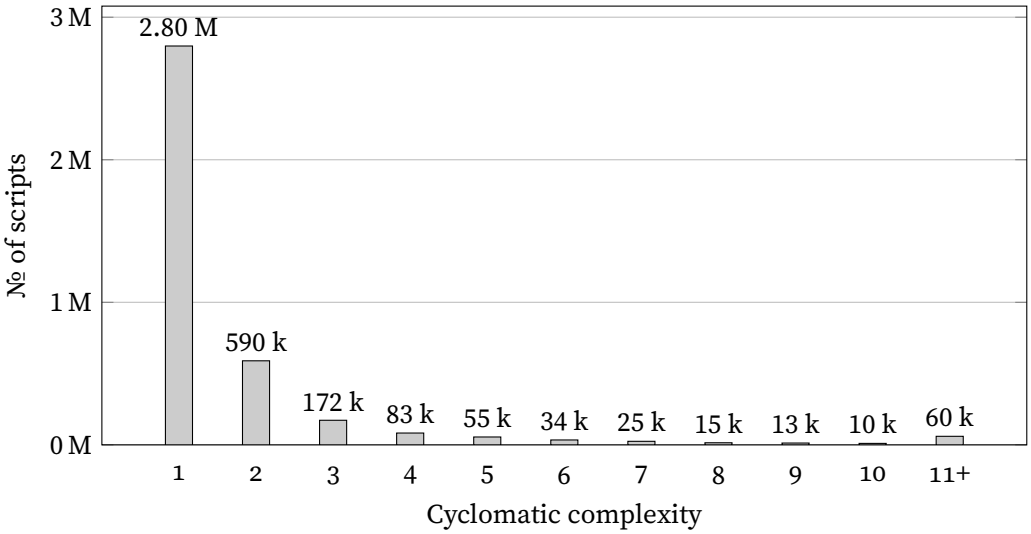


Figure 7.6. Distribution of scripts based on their cyclomatic complexity. Scripts with a complexity higher than 10 have been bundled into the last bucket.

Table 7.2. Prevalence of programming concepts in Scratch projects. The third column shows the results found by Aivaloglou and Hermans (2016) in percentage if available.

Concept	№ of projects	%	Aivaloglou et al. (%)
User input blocks	90 703	46.43	56.24
Random	67 403	34.50	N/A
Conditional statements	80 664	41.29	39.81
Loop statements	151 050	77.31	77.18
Repeat with condition	28 526	14.60	13.59
Variables	67 526	34.56	31.51
Lists	18 331	9.38	4.01
Procedures	33 319	17.05	7.70

7.6.6. Programming concepts

Table 7.2 is an overview of the prevalence of some programming concepts in the analysed Scratch projects. While the number of projects that use procedures (17.1 %, *7.70 %*) is higher, it is still not used that much: a majority of projects do not use it. Most projects do use loop statements (77.3 %, *77.18 %*), yet the number of projects using a conditional loop is much smaller (14.6 %, *13.59 %*). Less than half of projects use conditional statements (41.29 %, *39.81 %*), and a bit less than half (46.43 %, *56.24 %*) use user input blocks. About a third (34.56 %, *31.51 %*) of the projects use variables, and only 9.38 % (*4.01 %*) uses lists. This means that a large number of projects is simple (and this is what we would expect, given the previous metrics on project size and complexity).

In summary, most Scratch projects are small and simple. However, big and complex ones do exist. Scratch projects have not changed significantly since the analysis by Aivaloglou and Hermans (2016), even if Scratch 3.0 was released in that period. However, this was expected: Scratch 3.0 did not introduce major changes to Scratch-the-programming-language. The main differences are as follows: the number of complex projects and of projects with procedures increased slightly, while user input blocks are used a bit less.

7.7. Evaluation of the new execution model

To ascertain the effect of the new execution model on existing Scratch projects, we perform and report on a preliminary benchmark. We measure the performance and behaviour of various projects using the existing Scratch 3.0 execution model and variations of the new execution model.

We consider five variations of the new execution model, which differ in how fast they run: **EM-30** runs at 30 fps, **EM-60** at 60 fps, **EM-90** at 90 fps, **EM-120** at 120 fps, and **EM-ASAP** runs as fast as possible, meaning a new frame is started as soon as the previous frame finishes.

7.7.1. Selection of projects

First, we differentiate between large and small projects. In the previous analysis, we determined that 75 % of projects have less than 5 sprites, 11 scripts, and 80 blocks. We thus consider projects small if they have less than 80 blocks.

Secondly, we differentiate between projects with user interaction and those without user interaction. While 46.43 % of projects require user interaction, this makes those projects much more difficult to automatically benchmark. We therefore only manually look at two such projects.

7.7.2. Non-interactive projects

The benchmark dataset for non-interactive projects consists of 100 randomly chosen projects from the analysis dataset. Of those 100 projects, 75 are small projects. For these projects, we measure the number of executed blocks. This counts the number of times every block (from the physical lines of code, so excluding arguments) is executed. The benchmark dataset contains both projects that end and those that do not. Projects that do not end are, for example, those with repeat forever blocks. For projects that end, we measure the total number of executed blocks. Non-ending projects are halted after 60 s, and the number of executed blocks within that time is counted.

Figures 7.7a and 7.7b show how many projects differ in behaviour for small and large projects respectively. Figures 7.7c and 7.7d then show how much the projects differ. The results show that most projects do not differ a lot, but there are a few outliers, particularly with the EM-ASAP model, and with larger projects. In general, the EM-90 or EM-120 model seems to provide the most similar behaviour for most projects.

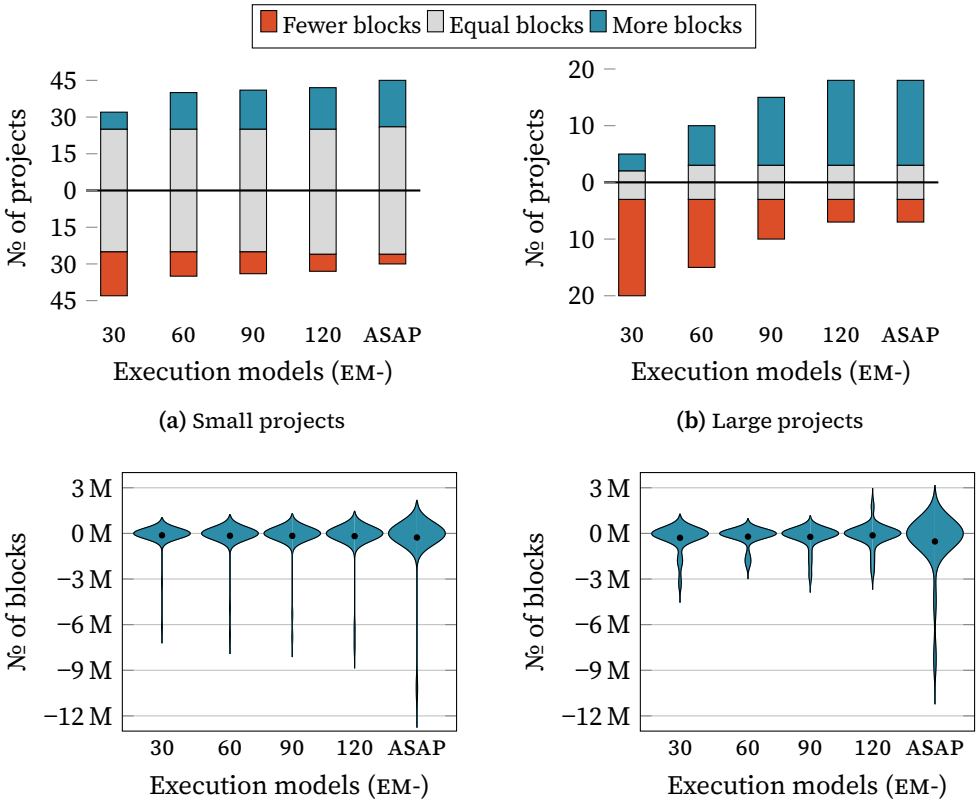
However, the behaviour of the EM-ASAP model warrants future research. Our initial investigation of those outlier projects did not reveal an immediate cause why the behaviour is so different. One hypothesis is that this is due to how the models are implemented: there are some quirks with JavaScript's `setTimeout` function, which is what is used in the virtual machine. An alternative implementation of the EM-ASAP model might be warranted.

7.7.3. Interactive projects

Lightning project

The first project we looked at in detail is the *Lightning* project (figure 7.8a).[5] The aim of this game is to dodge lightning bolts coming out of the sky. Figure 7.9a gives an overview of the behaviour with the different execution models. The EM-30,EM-60, and EM-90 model did not achieve 10 points: they were too slow and made the game unplayable.

[5] `https://scratch.mit.edu/projects/995927372/`



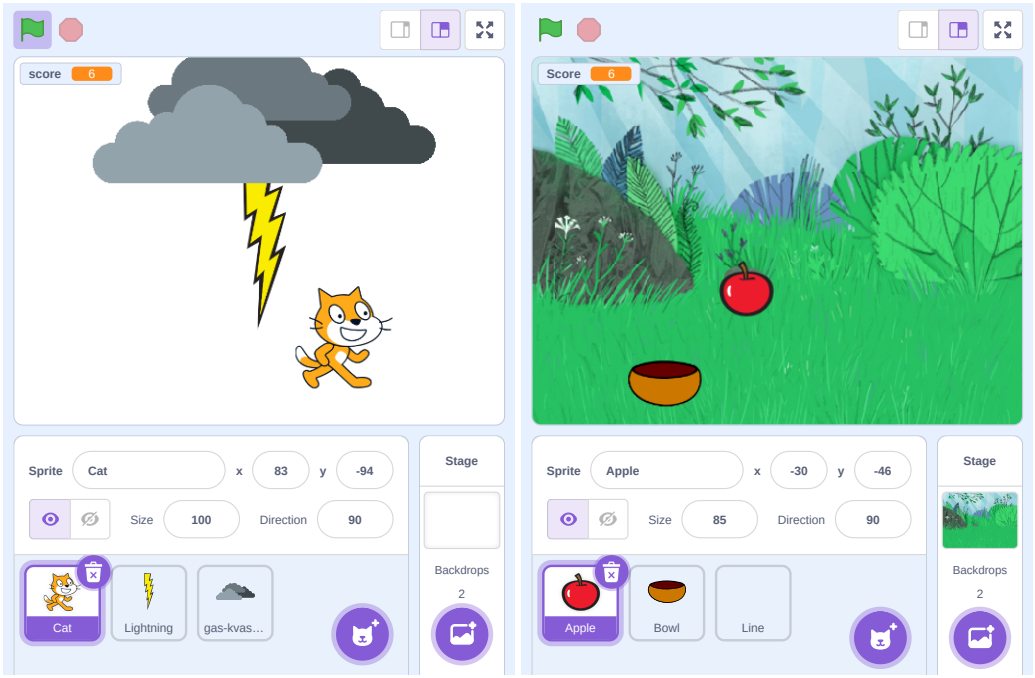
(a) Small projects

(b) Large projects

(c) Small projects

(d) Large projects

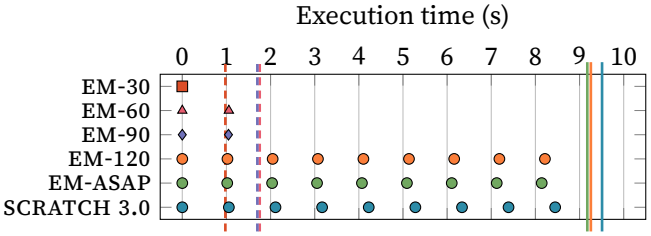
Figure 7.7. Variants of the new execution model compared against the original Scratch 3.0 execution model. The top figures show the number of projects that execute more, equal, or fewer blocks than the original execution model. The bottom figures show the difference in the number of block executions compared to the original execution model. A negative number indicates that fewer blocks were executed than in the original execution model.



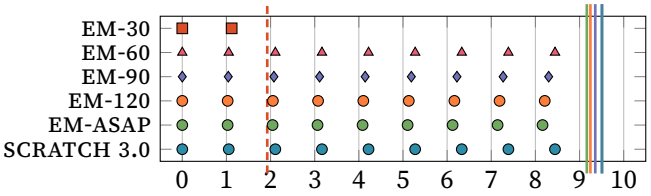
(a) The *Lightning* project in action. The user must use the arrow keys to move the cat around, avoiding the lightning bolts.

(b) The *Catch the apples* project in action. The basket follows the mouse, and the user must catch as many apples as possible, without any apples falling on the ground.

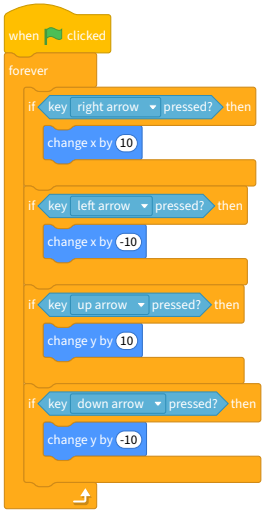
Figure 7.8. Overview of the two games we discuss.



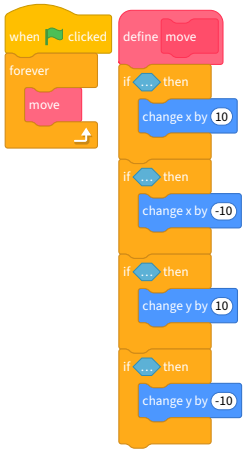
(a) The original implementation (see bottom left). The EM-30,EM-60, and EM-90 did not achieve 10 points.



(b) A modified implementation (see bottom right)



(c) Original implementation of the interactive component.



(d) Alternative implementation of the interactive component. The four if blocks have been moved to a procedure "run without screen refresh".

Figure 7.9. Behaviour and implementation of the *Lightning* exercise with different execution models. In the behaviour (top), marks represent when a new clone is made, while vertical lines indicate when the player either lost (dashed line) or completed 10 points (full line). The implementation (bottom) shows both variants of the user interaction code.

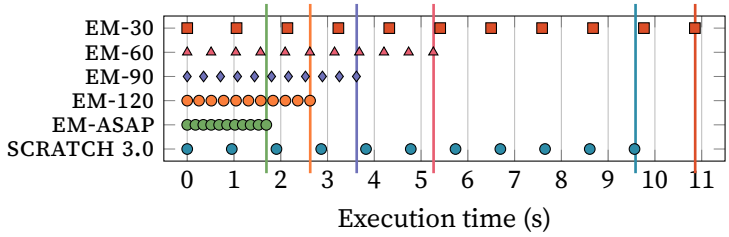


Figure 7.10. Behaviour of the *Catch the apples* exercise with different execution models. Marks represent score changes (or the initial score at 0), while vertical lines indicate when the player hits 10 points.

This is due to how the interactive component is implemented (figure 7.9c). Adjusting the implementation by wrapping the user interaction blocks with a procedure (figure 7.9d) enabled solves this issue (figure 7.9b). This implementation is usable with all new execution models, except EM-30, which remains too slow.

Catch the apples project

The second project is the *Catch the apples* project (figure 7.8b).[6] The aim here is to use the mouse to move a basket and catch as many falling apples as possible. Figure 7.10 shows the behaviour with different execution models. In this exercise, all models are playable, with EM-30 behaving the most similar to the original Scratch execution model. Models faster than EM-60 are likely too fast to actually play.

7.7.4. Discussion

A few conclusions can be drawn from the results of this preliminary benchmark.

First, it is not obviously clear which variation of the execution model is the best universal replacement. Different types of projects have different needs. One solution for this problem would be to modify the virtual machine to change the frames per second depending on the type of project that is executed.

Secondly, projects depend on specific behaviour of the virtual machine. For example, the *Lightning* project's implementation of user interaction only works because of how the current execution model works. While maybe unfortunate, as there are alternatives that do not depend on this

[6] `https://scratch.mit.edu/projects/995778768/`

behaviour, this is a consequence of *Hyrum's Law*, which states that "With a sufficient number of users, [...] all observable behaviours of your system will be depended on by somebody". Users, however, are already accustomed to changing the duration of wait blocks (and other such blocks) to account for the performance of the Scratch virtual machine on their device. While not ideal, small differences may thus be acceptable.

Lastly, this illustrates the need for more research into automated evaluation of the behaviour exhibited by Scratch projects. Some areas of interest are determining what constitutes observable behaviour and when changes to the behaviour become adverse changes. For example, if projects are executed a bit faster or slower, this might not affect the project's usability. The existing virtual machine is also not deterministic and depends on system performance: some projects might execute a lot slower on slower hardware, while still being usable.

7.8. Impact and conclusion

In chapter 6, we proposed a debugger for Scratch with a non-traditional step method. Instead of stepping a single block at a time, we want to step a single block in every thread of a Scratch program (thus in every script). However, this introduces two downsides: *i*) the step function exposes internal program state that is normally not visible to the users, and *ii*) the debugger uses a different execution model compared to regular execution, whereas a debugger should deviate from normal execution as little as possible.

This chapter then asks if we can modify the execution model of Scratch in such a way that the step functionality of a debugger is possible and that the changed execution model is usable for normal execution. To this end, we first took a detailed look at the existing execution model, due to a lack of existing literature on the topic. This model is the result of multiple years of work in the Scratch virtual machine and contains many nuances. The current execution model has been chosen to avoid some race conditions but does not avoid all concurrency-related issues: there are some surprising consequences of the threading model in particular.

Any changes to the execution model must not have adverse consequences for existing Scratch projects. Therefore, we first analyse how Scratch is used by replicating select metrics of previous investigations into what Scratch projects look like. The most significant previous result in this area is Aivaloglou and Hermans (2016), which analyses Scratch 2.0 projects.

Our results for Scratch 3.0 projects are broadly similar: most Scratch projects are small and simple, but there are a few big and complex ones.

Our proposed changes to the execution model change the threading from cooperative to preemptive. This implies that more race conditions, which the original execution model sought to prevent, are now possible. Maloney et al. (2010) use the classic example of reading the value of a variable, increasing this value, and finally updating the variable with the new value. When running the same code unchanged in the new execution model, more race conditions are possible. However, we believe this is not a big problem, due to two reasons. First, the kind of code where these race conditions can occur is used infrequently in Scratch.

Second, there is a workaround: the part of the code that must be protected against these race conditions can be extracted into a procedure, using the option "run without screen refresh". This causes the code in the function to become atomic: it will be run without interruptions. Using this technique is our recommendation for implementing critical sections. Note that using these critical sections with blocks that require a redraw should be done thoughtfully, as the redraw will not occur (this behaves identically in the current execution model).

Finally, we performed a preliminary benchmark using our proposed changes on a selection of projects, informed by our previous analysis. The results of the benchmark make clear that finding a single replacement execution model without affecting existing projects is highly unlikely. Projects depend on the behaviour of the current execution model, meaning any change in behaviour will be a breaking change. Considering the Scratch Team's (understandable) reluctance to introduce behavioural changes to Scratch at this point, we do not envision our changes being upstreamed, nor would we recommend it, unless as a breaking change.

Scratch 4.0?

While a universally applicable replacement execution model is not achievable, our proposed changes are still useful. For small projects, the impact of the changes is acceptable, and most projects are small. Additionally, the new execution model is more suitable for use with our stepping method for debuggers. We thus envision the new execution model to be used in classrooms where the debugger is used as well.

The results of the benchmarks are also a preliminary exploration. Our proposed changes must still be validated in educational practice and in a classroom setting with actual users of Scratch, in additional to performing expanded automated benchmarks. Another area of improvement is looking at more projects with user interaction for benchmarking. One possible route is investigating heuristics to automatically provide suitable user interaction to those projects, making them suitable for automated

benchmarking. Improving the benchmarking allows for a better understanding on the impact of the proposed execution model.

We also see more opportunities for research on the existing and new execution model of Scratch itself. For example, the current execution model is defined by its source code. Constructing a formal mode of the execution model would allow formal reasoning and analysis, which might reveal more opportunities for changes. This might also be beneficial in further analysing the new model's impact on existing Scratch projects.

Chapter 8.

Conclusions and opportunities

As an answer to the five research questions mentioned in section 1.3, we introduced five educational tools to facilitate programming education: two for textual programming languages and three for block-based programming languages. We also propose changes to the Scratch execution model.

We discuss each research question in detail below.

8.1. Textual programming languages

RQ1 Can we design an educational software testing framework that supports automated assessment across programming languages based on a single test suite?

We can, as we demonstrate by presenting our implementation of such a testing framework: **TESTed**. First, we identified input/output testing and unit testing as two opposing strategies commonly used in educational software testing. Our initial investigation focused on understanding how these approaches affect the supported programming languages within the testing frameworks. Often, testing frameworks that fall under input/output testing support multiple programming languages, but the quality of the feedback suffers. On the other hand, frameworks using unit testing have much more fine-grained feedback, but only support a single programming language.

Our aim was to combine the best of both worlds. To this end, we formulated the requirements for programming-language-agnostic testing frameworks that combine unit testing with support for multiple programming languages. We then introduced TESTed, and detailed its internal workings. Finally, we evaluated TESTed in educational practice to verify

that it supports our requirements for a programming-language-agnostic testing framework.

We see opportunities for more work on TESTed in the future. Our goal is to further develop TESTed for authoring different types of programming exercises across programming languages. TESTed is currently focused mainly on dynamic testing. A key area of future interest is the implementation of language-agnostic static code analysis capabilities.

RQ2 What is the most ergonomic way to author programming exercises with support for automated assessment across programming languages?

We concluded that a domain-specific language, designed specifically for this purpose, is the best approach. We then introduced our implementation: **TESTed-DSL**.

We again first looked at input/output testing and unit testing as the two opposing strategies. However, this time, we focused more on the impact of these strategies on the testing process itself. We considered what can be tested and how, in addition to how and what feedback is generated. For example, we considered if input/output testing and unit testing each needed a separate domain-specific language, or if we could merge them into one common one (we did merge them).

The conclusion was again that the best of both strategies provides the best experience for educators. Looking at programming-language-agnostic testing frameworks more broadly, we reported on three benefits for the adoption of such frameworks: *i*) sharing the same declarative structure across programming languages, *ii*) bridging the gap between input/output testing and unit testing, and *iii*) allowing test code to be expressed in a language-agnostic way.

We also see potential for additions to TESTed-DSL in the future. These include supporting operator overloading, string conversion, comments, indexing sequences, indexing mappings, destructuring, object identity checking, and object equivalence checking. Native support for pretty printing nested data structures would be another valuable addition, making it easier to detect differences between expected and actual return values. There are more opportunities still, including data-driven tests (parameterized tests), supporting dynamic generation of test data and boosting the performance of running tests.

8.2. Block-based programming languages

RQ3 Can we design an educational software testing framework for the block-based programming language Scratch?

Yes, as shown by our implementation of such a framework: **Itch**. We showed that it offers a versatile approach to testing, allowing static testing, emulating user interaction, and performing post-mortem testing. Tests can range from purely static to purely dynamic, or a hybrid of both.

However, we also reported that while most exercises can be tested, it remains difficult to design Scratch exercises that are both dynamically testable and sufficiently open-ended to align with the game-like and exploratory nature of Scratch. Static tests, though faster and sometimes easier to write, can potentially constrain creativity and go against the spirit of Scratch.

We also found that educators that are primarily experienced in Scratch might find JavaScript test suites difficult to write. For this reason, we created and reported on a prototype of a Scratch-based testing framework called **Poke**. It allows creating test suites with Scratch, using the blocks and environment Scratch users are familiar with. While writing the tests is technically feasible, some challenges remain, the main one being the organizational aspects of managing these Scratch tests suites.

RQ4 Can we design an (educational) debugger for the block-based programming language Scratch?

Yes: **Blink** is our time-travelling debugger for Scratch. Working with Itch in an educational setting made clear that testing frameworks primarily indicate whether a submission is correct or not, but do not directly assist students in finding the root cause of a failed test.

To address this, we developed Blink, a debugger for Scratch. Debuggers are generally known to be good tools for finding errors in a program, and this is no different in Scratch. Blink supports pausing execution, stepping through code, breakpoints, and provides time travel capabilities. We prioritized the user-friendliness of Blink, given Scratch's younger target audience. Due to the concurrent nature of Scratch, we had to hide a lot of the complexities of concurrent debugging for the user. Initial feedback from using Blink in a classroom setting has been positive: students find the debugger intuitive and useful, especially the time-travelling capability.

In the future, we envision integrating Itch with Blink. In an ideal scenario, a failed test from Itch would allow students to directly open a debugging

session when the test failed. Using the time-travelling features of Blink, students can then go back in time until they find the issue.

RQ5 What common execution model for running and debugging Scratch code best optimizes both scenarios?

One of the ways we sought to make the debugger more intuitive was to use a non-traditional stepping functionality. In most debuggers, a step will advance the code one step in a single thread. However, in Scratch, we wanted the step to advance one step in all threads simultaneously. The current execution model makes this difficult.

To answer this research question, we first explained in detail how the current execution model behaves. The current execution model has been chosen to minimize the occurrence of some concurrency-related issues, like certain race conditions, but does not prevent all issues. The threading model, in particular, causes some surprising behaviour, which is not ideal as Scratch is intended to be intuitive.

Next, we proposed changes to the execution model of Scratch that seek to resolve these issues. However, the widespread use of Scratch demands that any changes must not adversely affect existing projects. For this reason, we explored how Scratch is used, the results of which corroborate previous findings: most Scratch projects are small.

We then performed a preliminary benchmark of the new execution model on various representative projects to measure the real-world impact. The results of this benchmark make clear that our initial goal of finding a replacement execution model that has no effect on existing projects is not attainable. Projects rely too much on the behaviour of the original execution model: any change to this behaviour will be a breaking change. However, the benchmark also showed that the effect on smaller projects is acceptable, and most Scratch projects are small. We thus believe the new execution model to still be useful: not as a general replacement, but for use in specific contexts. For example, in a classroom setting, we believe using the new execution model improves the experience of using the debugger.

Finally, we believe more research is needed in regard to the execution model of Scratch. For starters, the execution model is currently defined by its source code, despite some attempts to create a formal model for it. As an example, constructing full operational semantics for the execution model would allow better formal reasoning and analysis, which might reveal more opportunities for changes. Secondly, the changes we proposed in this chapter should be validated experimentally in a classroom setting. This would allow verification if the proposed changes are intuitive.

Bibliography

Agrawal, A. and B. Reed (Nov. 2022). "A Survey on Grading Format of Automated Grading Tools for Programming Assignments". In: *International Conference of Education, Research and Innovation Proceedings*. Seville, Spain: IATED, pp. 7506–7514. ISBN: 978-84-09-45476-1. DOI: 10.21125/iceri.2022.1912.

Aivaloglou, E. and F. Hermans (Aug. 2016). "How Kids Code and How We Know: An Exploratory Study on the Scratch Repository". In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*. Melbourne, Australia: ACM, pp. 53–61. ISBN: 978-1-4503-4449-4. DOI: 10.1145/2960310.2960325.

Ala-Mutka, K. M. (June 2005). "A Survey of Automated Assessment Approaches for Programming Assignments". In: *Computer Science Education* 15.2, pp. 83–102. ISSN: 0899-3408. DOI: 10.1080/08993400500150747.

AlOmar, E. A., S. A. AlOmar and M. W. Mkaouer (May 2023). "On the Use of Static Analysis to Engage Students with Software Quality Improvement: An Experience with PMD". In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training*. Melbourne, Australia: IEEE, pp. 179–191. ISBN: 9798350322590. DOI: 10.1109/ICSE-SEET58685.2023.00023.

Ammann, P. and J. Offutt (Dec. 2016). *Introduction to Software Testing*. 2nd ed. Cambridge University Press. ISBN: 978-1-107-17201-2. DOI: 10.1017/9781316771273.

Atchison, W. F. (Apr. 1971). "Computer Science as a New Discipline". In: *The International Journal of Electrical Engineering & Education* 9.2, pp. 130–135. DOI: 10.1177/002072097100900209.

Balanskat, A. and K. Engelhardt (Oct. 2015). *Computing Our Future. Computer Programming and Coding Priorities, School Curricula and Initiatives across Europe.* Brussels, Belgium: European Schoolnet, p. 45. URL: http://www.eun.org/news/detail?articleId=652951.

Ball, T., A. Chatra, P. de Halleux, S. Hodges, M. Moskal and J. Russell (Oct. 2019). "Microsoft MakeCode: Embedded Programming for Education, in Blocks and TypeScript". In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. Athens, Greece: ACM, pp. 7–12. ISBN: 978-1-4503-6989-3. DOI: 10.1145/3358711.3361630.

Balzer, R. M. (May 1969). "EXDAMS: Extendable Debugging and Monitoring System". In: *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference (AFIPS '69)*. Boston, USA: ACM, pp. 567–580. ISBN: 978-1-4503-7902-1. DOI: 10.1145/1476793.1476881.

Barr, E. T. and M. Marron (Dec. 2014). "Tardis: Affordable Time-Travel Debugging in Managed Runtimes". In: *ACM SIGPLAN Notices* 49.10, pp. 67–82. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2714064.2660209.

Barr, E. T., M. Marron, E. Maurer, D. Moseley and G. Seth (Nov. 2016). "Time-Travel Debugging for JavaScript/Node.Js". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Seattle, USA: ACM, pp. 1003–1007. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2983933.

Bass, L., P. Clements and R. Kazman (2021). *Software Architecture in Practice*. 4th ed. Addison-Wesley Professional. ISBN: 978-0-13-688567-2.

Bau, D., J. Gray, C. Kelleher, J. Sheldon and F. Turbak (May 2017). "Learnable Programming: Blocks and Beyond". In: *Communications of the ACM* 60.6, pp. 72–80. ISSN: 0001-0782. DOI: 10.1145/3015455.

Beck, K. (1997). "Simple Smalltalk Testing". In: *Kent Beck's Guide to Better Smalltalk: A Sorted Collection*. Cambridge, UK: Cambridge University Press, pp. 277–288. ISBN: 978-0-511-57497-9. DOI: 10.1017/CBO9780511574979.

Bejaković, P. and Ž. Mrnjavac (Apr. 2020). "The Importance of Digital Literacy on the Labour Market". In: *Employee Relations: The International Journal* 42.4, pp. 921–932. ISSN: 0142-5455. DOI: 10.1108/ER-07-2019-0274.

Ben-Kiki, O., T. Müller, I. döt Net, P. Antoniou, E. Aro, T. Smith and C. C. Evans (Oct. 2021). *YAML Ain't Markup Language*. URL: https://yaml.org/spec/1.2.2/.

Berssanette, J. H. and A. C. de Francisco (2021). "Active Learning in the Context of the Teaching/Learning of Computer Programming: A Systematic Review". In: *Journal of Information Technology Education: Research* 20, pp. 201–220. ISSN: 1547-9714, 1539-3585. DOI: 10.28945/4767.

Bettini, L., P. Crescenzi, G. Innocenti, M. Loreti and L. Cecchi (2004). "An Environment for Self-Assessing Java Programming Skills in Undergraduate First Programming Courses". In: *IEEE International Conference on Advanced Learning Technologies, 2004. Proceedings.* Joensuu, Finland: IEEE, pp. 161–165. ISBN: 978-0-7695-2181-7. DOI: 10.1109/ICALT.2004.1357395.

Bez, J. L., N. A. Tonin and P. R. Rodegheri (Aug. 2014). "URI Online Judge Academic: A Tool for Algorithms and Programming Classes". In: *2014 9th International Conference on Computer Science & Education*. Vancouver, Canada: IEEE, pp. 161–165. ISBN: 978-1-4799-2951-1. DOI: 10.1109/iccse.2014.6926445.

Bissyande, T. F., F. Thung, D. Lo, L. Jiang and L. Reveillere (July 2013). "Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects". In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. Kyoto, Japan: IEEE, pp. 303–312. ISBN: 978-0-7695-4986-6. DOI: 10.1109/COMPSAC.2013.55.

Black, P. and D. Wiliam (Feb. 2009). "Developing the Theory of Formative Assessment". In: *Educational Assessment, Evaluation and Accountability* 21.1, pp. 5–31. ISSN: 1874-8597. DOI: 10.1007/s11092-008-9068-5.

Boe, B., C. Hill, M. Len, G. Dreschler, P. Conrad and D. Franklin (Mar. 2013). "Hairball: Lint-inspired Static Analysis of Scratch Projects". In: *SIGCSE '13: Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. Denver, USA: ACM, pp. 215–220. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445265.

Booker, A. R. (July 2019). "Cracking the Problem with 33". In: *Research in Number Theory* 5.3, p. 26. ISSN: 2363-9555. DOI: 10.1007/s40993-019-0162-1.

Bunce, T. (Apr. 2008). *TIOBE or Not TIOBE – "Lies, Damned Lies, and Statistics"*. Not this... URL: https://blog.timbunce.org/2008/04/12/tiobe-or-not-tiobe-lies-damned-lies-and-statistics/.

Bunce, T. (May 2009). *TIOBE Index Is Being Gamed*. Not this... URL: https://blog.timbunce.org/2009/05/17/tiobe-index-is-being-gamed/.

Caiza, J. C. and J. M. del Alamo (2013). "Programming Assignments Automatic Grading: Review of Tools and Implementations". In: *INTED2013 Proceedings*. 7th International Technology, Education and Development Conference. Valencia, Spain: IATED,

pp. 5691–5700. ISBN: 978-84-616-2661-8. URL: https://library.iated.org/view/CAIZA2013PRO.

Camp, T., W. R. Adrion, B. Bizot, S. Davidson, M. Hall, S. Hambrusch, E. Walker and S. Zweben (May 2017). "Generation CS: The Growth of Computer Science". In: *ACM Inroads* 8.2, pp. 44–50. ISSN: 2153-2184. DOI: 10.1145/3084362.

Campos, D. S., A. J. Mendes, M. J. Marcelino, D. J. Ferreira and L. M. Alves (Oct. 2012). "A Multinational Case Study on Using Diverse Feedback Types Applied to Introductory Programming Learning". In: *2012 Frontiers in Education Conference Proceedings*. Seattle, USA: IEEE, pp. 1–6. ISBN: 978-1-4673-1351-3. DOI: 10.1109/FIE.2012.6462412.

Cattoire, H., P. Dawyndt, C. Scholliers and N. Strijbol (2024). "Een nieuw uitvoeringsmodel voor Scratch 3.0". MA thesis. Universiteit Gent.

Cavalcanti, A. P., A. Barbosa, R. Carvalho, F. Freitas, Y.-S. Tsai, D. Gašević and R. F. Mello (2021). "Automatic Feedback in Online Learning Environments: A Systematic Literature Review". In: *Computers and Education: Artificial Intelligence* 2, p. 100027. ISSN: 2666920X. DOI: 10.1016/j.caeai.2021.100027.

Cheang, B., A. Kurnia, A. Lim and W.-C. Oon (Sept. 2003). "On Automated Grading of Programming Assignments in an Academic Institution". In: *Computers & Education* 41.2, pp. 121–131. ISSN: 03601315. DOI: 10.1016/S0360-1315(03)00030-7.

Chen, S.-K., W. K. Fuchs and J.-Y. Chung (Aug. 2001). "Reversible Debugging Using Program Instrumentation". In: *IEEE Transactions on Software Engineering* 27.8, pp. 715–727. ISSN: 0098-5589. DOI: 10.1109/32.940726.

Cherf, G. S. (Sept. 1992). "An Investigation of the Maintenance and Support Characteristics of Commercial Software". In: *Software Quality Journal* 1.3, pp. 147–158. ISSN: 0963-9314, 1573-1367. DOI: 10.1007/BF01720922.

Chung, M. J.-Y., M. Nakura, S. H. Neti, A. Lu, E. Hummel and M. Cakmak (Aug. 2020). "ConCodeIt! A Comparison of Concurrency Interfaces in Block-Based Visual Robot Programming". In: *2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*. Naples, Italy: IEEE, pp. 245–252. ISBN: 978-1-72816-075-7. DOI: 10.1109/RO-MAN47096.2020.9223337.

Combéfis, S. (Feb. 2022). "Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools". In: *Software* 1.1, pp. 3–30. ISSN: 2674-113X. DOI: 10.3390/software1010002.

Crescenzi, P., C. Demetrescu, I. Finocchi and R. Petreschi (2000). "Reversible Execution and Visualization of Programs with LEONARDO". In: *Journal of Visual Languages & Computing* 11.2, pp. 125–150. ISSN: 1045-926X. DOI: 10.1006/jvlc.1999.0143.

Czaplicki, E. and S. Chong (June 2013). "Asynchronous Functional Reactive Programming for GUIs". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Seattle, USA: ACM, pp. 411–422. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462161.

De Proft, R., P. Dawyndt, C. Scholliers and N. Strijbol (2022). "Blink: een educatieve software-debugger voor Scratch 3.0". MA thesis. Universiteit Gent. URL: http://lib.ugent.be/catalog/rug01:003059967.

Deiner, A., P. Feldmeier, G. Fraser, S. Schweikl and W. Wang (May 2023). "Automated Test Generation for Scratch Programs". In: *Empirical Software Engineering* 28.3, p. 79. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-022-10255-x.

Deiner, A., C. Frädrich, G. Fraser, S. Geserer and N. Zantner (Oct. 2020). "Search-Based Testing for Scratch Programs". In: *Proceedings of the 12th International Symposium on*

Search-Based Software Engineering. Ed. by A. Aleti and A. Panichella. Vol. 12420. Bari, Italy: Springer, pp. 58–72. ISBN: 978-3-030-59762-7. DOI: 10.1007/978-3-030-59762-7_5.

Deiner, A. and G. Fraser (Feb. 2024). "NuzzleBug: Debugging Block-Based Programs in Scratch". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. Lisbon, Portugal: ACM, pp. 1–13. ISBN: 9798400702174. DOI: 10.1145/3597503.3623331.

Denning, P. J. (May 2013). "The Science in Computer Science". In: *Communications of the ACM* 56.5, pp. 35–38. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2447976.2447988.

De Souza, D. M., K. R. Felizardo and E. F. Barbosa (Apr. 2016). "A Systematic Literature Review of Assessment Tools for Programming Assignments". In: *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. Dallas, USA: IEEE, pp. 147–156. ISBN: 978-1-5090-0766-0. DOI: 10.1109/CSEET.2016.48.

Douce, C., D. Livingstone and J. Orwell (Sept. 2005). "Automatic Test-Based Assessment of Programming: A Review". In: *Journal on Educational Resources in Computing* 5.3, p. 4. ISSN: 1531-4278, 1531-4278. DOI: 10.1145/1163405.1163409.

Edwards, S. H. (Mar. 2004). "Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action". In: *ACM SIGCSE Bulletin* 36.1, pp. 26–30. ISSN: 0097-8418. DOI: 10.1145/971300.971312.

Edwards, S. H., J. Börstler, L. N. Cassel, M. S. Hall and J. Hollingsworth (Nov. 2008). "Developing a Common Format for Sharing Programming Assignments". In: *ACM SIGCSE Bulletin* 40.4, pp. 167–182. ISSN: 0097-8418. DOI: 10.1145/1473195.1473240.

Ellsworth, C. C., J. B. Fenwick and B. L. Kurtz (Mar. 2004). "The Quiver System". In: *ACM SIGCSE Bulletin* 36.1, pp. 205–209. ISSN: 0097-8418. DOI: 10.1145/1028174.971374.

Enstrom, E., G. Kreitz, F. Niemela, P. Soderman and V. Kann (Oct. 2011). "Five Years with Kattis — Using an Automated Assessment System in Teaching". In: *2011 Frontiers in Education Conference (FIE)*. Rapid City, USA: IEEE, T3J-1-T3J–6. ISBN: 978-1-61284-467-1. DOI: 10.1109/FIE.2011.6142931.

Fatourou, E., N. C. Zygouris, T. Loukopoulos and G. I. Stamoulis (June 2018). "Teaching Concurrent Programming Concepts Using Scratch in Primary School: Methodology and Evaluation". In: *International Journal of Engineering Pedagogy (iJEP)* 8.4, p. 89. ISSN: 2192-4880. DOI: 10.3991/ijep.v8i4.8216.

Fenton, N. E. and M. Neil (1999). "A Critique of Software Defect Prediction Models". In: *IEEE Transactions on Software Engineering* 25.5, pp. 675–689. ISSN: 00985589. DOI: 10.1109/32.815326.

Fonte, D., D. da Cruz, A. L. Gançarski and P. R. Henriques (June 2013). "A Flexible Dynamic System for Automatic Grading of Programming Exercises". In: *2nd Symposium on Languages, Applications and Technologies*. Ed. by J. P. Leal, R. Rocha and A. Simões. Vol. 29. OpenAccess Series in Informatics (OASIcs). Porto, Portugal: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 129–144. ISBN: 978-3-939897-52-1. DOI: 10.4230/OASIcs.SLATE.2013.129.

Frädrich, C., F. Obermüller, N. Körber, U. Heuer and G. Fraser (June 2020). "Common Bugs in Scratch Programs". In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. Trondheim, Norway: ACM, pp. 89–95. ISBN: 978-1-4503-6874-2. DOI: 10.1145/3341525.3387389.

Fraser, G., U. Heuer, N. Körber, F. Obermüller and E. Wasmeier (May 2021). "LitterBox: A Linter for Scratch Programs". In: *2021 IEEE/ACM 43rd International Conference on*

Software Engineering: Software Engineering Education and Training. Madrid, Spain: IEEE, pp. 183–188. ISBN: 978-1-66540-138-8. DOI: 10.1109/ICSE-SEET52601.2021.00028.
Fronza, I., L. Corral and C. Pahl (Mar. 2020). "An Approach to Evaluate the Complexity of Block-Based Software Product". In: *Informatics in Education* 19.1, pp. 15–32. ISSN: 1648-5831, 2335-8971. DOI: 10.15388/infedu.2020.02.
Goethals, K., P. Dawyndt, C. Scholliers and N. Strijbol (2023). "Een Time Travelling Debugger Voor Scratch 3.0". MA thesis. Universiteit Gent. URL: http://lib.ugent.be/catalog/rug01:003150086.
Gomes, A. and A. J. Mendes (June 2007). "An Environment to Improve Programming Education". In: *Proceedings of the 2007 International Conference on Computer Systems and Technologies*. Ruse, Bulgaria: ACM, p. 1. ISBN: 978-954-9641-50-9. DOI: 10.1145/1330598.1330691.
Gorn, S. (Apr. 1963). "The Computer and Information Sciences: A New Basic Discipline". In: *SIAM Review* 5.2, pp. 150–155. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/1005036.
Gotz, K., P. Feldmeier and G. Fraser (Apr. 2022). "Model-Based Testing of Scratch Programs". In: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Valencia, Spain: IEEE, pp. 411–421. ISBN: 978-1-66546-679-0. DOI: 10.1109/ICST53961.2022.00047.
Gulwani, S., I. Radiček and F. Zuleger (Nov. 2014). "Feedback Generation for Performance Problems in Introductory Programming Assignments". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, pp. 41–51. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635912.
Gupta, S. K. and B. B. Gupta (Jan. 2017). "Cross-Site Scripting (XSS) Attacks and Defense Mechanisms: Classification and State-of-the-Art". In: *International Journal of System Assurance Engineering and Management* 8.S1, pp. 512–530. ISSN: 0975-6809, 0976-4348. DOI: 10.1007/s13198-015-0376-0.
Gusukuma, L., A. C. Bart and D. Kafura (Feb. 2020). "Pedal: An Infrastructure for Automated Feedback Systems". In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. Portland, USA: ACM, pp. 1061–1067. ISBN: 978-1-4503-6793-6. DOI: 10.1145/3328778.3366913.
Halstead, M. H. (1977). *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier. ISBN: 0-444-00205-7.
Hamer, P. G. and G. D. Frewin (Sept. 1982). "M.H. Halstead's Software Science - a Critical Examination". In: *Proceedings of the 6th International Conference on Software Engineering*. Tokyo, Japan: IEEE, pp. 197–206. URL: https://dl.acm.org/doi/10.5555/800254.807762.
Hao, Q., D. H. Smith IV, L. Ding, A. Ko, C. Ottaway, J. Wilson, K. H. Arakawa, A. Turcan, T. Poehlman and T. Greer (Jan. 2021). "Towards Understanding the Effective Design of Automated Formative Feedback for Programming Assignments". In: *Computer Science Education*, pp. 1–23. ISSN: 0899-3408. DOI: 10.1080/08993408.2020.1860408.
Hattie, J. and H. Timperley (Mar. 2007). "The Power of Feedback". In: *Review of Educational Research* 77.1, pp. 81–112. ISSN: 0034-6543. DOI: 10.3102/003465430298487.
Hatton, L. (Aug. 2008). "Invited Talk: The Role of Empiricism in Improving the Reliability of Future Software". In: *Testing: Academic & Industrial Conference - Practice and Research Techniques (Taic Part 2008)*. Windsor, UK: IEEE. ISBN: 978-0-7695-3383-4. DOI: 10.1109/TAIC-PART.2008.21.

Hermans, F. and E. Aivaloglou (May 2016). "Do Code Smells Hamper Novice Programming? A Controlled Experiment on Scratch Programs". In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. Austin, USA: IEEE, pp. 1–10. ISBN: 978-1-5090-1428-6. DOI: 10.1109/ICPC.2016.7503706.

Hetzel, B. (Apr. 1988). *The Complete Guide to Software Testing*. 2nd ed. USA: QED Information Sciences. 280 pp. ISBN: 978-0-89435-242-3. URL: https://dl.acm.org/doi/book/10.5555/42384.

Hext, J. B. and J. W. Winings (May 1969). "An Automatic Grading Scheme for Simple Programming Exercises". In: *Communications of the ACM* 12.5, pp. 272–275. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/362946.362981.

Hidalgo-Céspedes, J. (Oct. 2023). "Evaluation of an Online Judge for Concurrent Programming Learning". In: *2023 XLIX Latin American Computer Conference (CLEI)*. La Paz, Bolivia: IEEE, pp. 1–9. ISBN: 9798350318876. DOI: 10.1109/CLEI60451.2023.10346201.

Higgins, C., T. Hegazy, P. Symeonidis and A. Tsintsifas (Sept. 2003). "The CourseMarker CBA System: Improvements over Ceilidh". In: *Education and Information Technologies* 8.3, pp. 287–304. ISSN: 1573-7608. DOI: 10.1023/A:1026364126982.

Hollingsworth, J. (Oct. 1960). "Automatic Graders for Programming Classes". In: *Communications of the ACM* 3.10, pp. 528–529. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/367415.367422.

Hopcroft, J. E. (1987). "Computer Science: The Emergence of a Discipline". In: *ACM Turing Award Lectures*. New York, NY, USA: ACM. ISBN: 978-1-4503-1049-9. DOI: 10.1145/1283920.1283943.

Howden, W. E. (July 1978). "Theoretical and Empirical Studies of Program Testing". In: *IEEE Transactions on Software Engineering* SE-4.4, pp. 293–298. ISSN: 0098-5589. DOI: 10.1109/TSE.1978.231514.

Hromkovič, J. and J. Staub (2021). "The Problem with Debugging in Current Block-Based Programming Environments". In: *Bulletin of the EATCS* 135.3. URL: http://smtp.eatcs.org/index.php/beatcs/article/view/667.

ICPC Fact Sheet (2023). ICPC. URL: https://icpc.global/worldfinals/fact-sheet/ICPC-Fact-Sheet.pdf.

Ihantola, P., T. Ahoniemi, V. Karavirta and O. Seppälä (Oct. 2010). "Review of Recent Systems for Automatic Assessment of Programming Assignments". In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli, Finland: ACM, pp. 86–93. ISBN: 978-1-4503-0520-4. DOI: 10.1145/1930464.1930480.

Johnson, D. E. (Feb. 2016). "ITCH: Individual Testing of Computer Homework for Scratch Assignments". In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. Memphis, USA: ACM, pp. 223–227. ISBN: 978-1-4503-3685-7. DOI: 10.1145/2839509.2844600.

Jones, D. (Apr. 2019). *Dimensional Analysis of the Halstead Metrics*. The Shape of Code. URL: https://shape-of-code.com/2019/04/25/dimensional-analysis-of-the-halstead-metrics/.

Kelleher, C. and R. Pausch (June 2005). "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers". In: *ACM Computing Surveys* 37.2, pp. 83–137. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/1089733.1089734.

Keuning, H., J. Jeuring and B. Heeren (Sept. 2018). "A Systematic Literature Review of Automated Feedback Generation for Programming Exercises". In: *ACM Transactions on Computing Education* 19.1, pp. 1–43. DOI: 10.1145/3231711.

Khorram, F. (Dec. 2022). "A Testing Framework for Executable Domain-Specific Languages". PhD thesis. Ecole nationale supérieure Mines-Télécom Atlantique. URL: https://theses.hal.science/tel-03977604.

Kim, C., J. Yuan, L. Vasconcelos, M. Shin and R. B. Hill (Oct. 2018). "Debugging during Block-Based Programming". In: *Instructional Science* 46.5, pp. 767–787. ISSN: 0020-4277, 1573-1952. DOI: 10.1007/s11251-018-9453-5.

Kim, H., H. Choi, J. Han and H.-J. So (Aug. 2012). "Enhancing Teachers' ICT Capacity for the 21st Century Learning Environment: Three Cases of Teacher Education in Korea". In: *Australasian Journal of Educational Technology* 28.6. ISSN: 1449-5554, 1449-3098. DOI: 10.14742/ajet.805.

Knuth, D. E. (Apr. 1974). "Computer Science and Its Relation to Mathematics". In: *The American Mathematical Monthly* 81.4, pp. 323–343. ISSN: 0002-9890, 1930-0972. DOI: 10.1080/00029890.1974.11993556.

Kosowski, A., M. Małafiejski and T. Noiński (2008). "Application of an Online Judge & Contester System in Academic Tuition". In: *Advances in Web Based Learning – ICWL 2007*. Ed. by H. Leung, F. Li, R. Lau and Q. Li. Vol. 4823. Berlin, Heidelberg: Springer, pp. 343–354. ISBN: 978-3-540-78138-7. DOI: 10.1007/978-3-540-78139-4_31.

Krusche, S. and A. Seitz (Feb. 2018). "ArTEMiS: An Automatic Assessment Management System for Interactive Learning". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. Baltimore, USA: ACM, pp. 284–289. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159602.

Kurnia, A., A. Lim and B. Cheang (May 2001). "Online Judge". In: *Computers & Education* 36.4, pp. 299–315. ISSN: 03601315. DOI: 10.1016/S0360-1315(01)00018-5.

Le, N.-T., F. Loll and N. Pinkwart (July 2013). "Operationalizing the Continuum between Well-Defined and Ill-Defined Problems for Educational Technology". In: *IEEE Transactions on Learning Technologies* 6.3, pp. 258–270. ISSN: 1939-1382. DOI: 10.1109/TLT.2013.16.

Leal, J. P. and F. Silva (May 2003). "Mooshak: A Web-based Multi-site Programming Contest System". In: *Software: Practice and Experience* 33.6, pp. 567–581. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.522.

Liu, K., Y. Han, J. M. Zhang, Z. Chen, F. Sarro, M. Harman, G. Huang and Y. Ma (July 2023). "Who Judges the Judge: An Empirical Study on Online Judge Tests". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Seattle, USA: ACM, pp. 334–346. ISBN: 9798400702211. DOI: 10.1145/3597926.3598060.

Luck, M. and M. Joy (July 1999). "A Secure On-Line Submission System". In: *Software: Practice and Experience* 29.8, pp. 721–740. ISSN: 1097-024X. DOI: 10.1002/(SICI)1097-024X(19990710)29:8<721::AID-SPE257>3.0.CO;2-0.

Luxton-Reilly, A., Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard and C. Szabo (July 2018). "Introductory Programming: A Systematic Literature Review". In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. Larnaca, Cyprus: ACM, pp. 55–106. ISBN: 978-1-4503-6223-8. DOI: 10.1145/3293881.3295779.

Maertens, R., M. Van Neyghem, M. Geldhof, C. Van Petegem, N. Strijbol, P. Dawyndt and B. Mesuere (May 2024). "Discovering and Exploring Cases of Educational Source Code

Plagiarism with Dolos". In: *SoftwareX* 26, p. 101755. ISSN: 23527110. DOI: 10.1016/j.softx.2024.101755.

Maertens, R., C. Van Petegem, N. Strijbol, T. Baeyens, A. C. Jacobs, P. Dawyndt and B. Mesuere (2022). "Dolos: Language-agnostic Plagiarism Detection in Source Code". In: *Journal of Computer Assisted Learning* 38.4, pp. 1046–1061. ISSN: 1365-2729. DOI: 10.1111/jcal.12662.

Mak, N., P. Dawyndt and C. Scholliers (2019). "Itch: een educatief testframework voor automatische feedback op Scratch projecten". MA thesis. Universiteit Gent. URL: http://lib.ugent.be/catalog/rug01:002782933.

Maloney, J., M. Resnick, N. Rusk, B. Silverman and E. Eastmond (Nov. 2010). "The Scratch Programming Language and Environment". In: *ACM Transactions on Computing Education* 10.4, pp. 1–15. ISSN: 1946-6226. DOI: 10.1145/1868358.1868363.

McCabe, T. J. (Dec. 1976). "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4, pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

McCauley, R., S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas and C. Zander (June 2008). "Debugging: A Review of the Literature from an Educational Perspective". In: *Computer Science Education* 18.2, pp. 67–92. ISSN: 0899-3408, 1744-5175. DOI: 10.1080/08993400802114581.

Messer, M., N. C. C. Brown, M. Kölling and M. Shi (Mar. 2024). "Automated Grading and Feedback Tools for Programming Education: A Systematic Review". In: *ACM Transactions on Computing Education* 24.1, pp. 1–43. ISSN: 1946-6226. DOI: 10.1145/3636515.

Meszaros, G. (May 2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley. 944 pp. ISBN: 978-0-13-149505-0.

Mishra, D. S. and S. H. Edwards (Mar. 2023). "The Programming Exercise Markup Language: Towards Reducing the Effort Needed to Use Automated Grading Tools". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. Toronto, Canada: ACM, pp. 395–401. ISBN: 978-1-4503-9431-4. DOI: 10.1145/3545945.3569734.

Mönig, J. and B. Harvey (Apr. 2024). *Snap! Build Your Own Blocks*. Version v9.2.17. URL: https://snap.berkeley.edu/.

Moreno-León, J. and G. Robles (Sept. 2015). "Dr. Scratch: A Web Tool to Automatically Evaluate Scratch Projects". In: *Proceedings of the 10th Workshop in Primary and Secondary Computing Education*. London, UK: ACM, pp. 132–133. ISBN: 978-1-4503-3753-3. DOI: 10.1145/2818314.2818338.

Murphy, E., T. Crick and J. H. Davenport (Apr. 2017). "An Analysis of Introductory Programming Courses at UK Universities". In: *The Art, Science, and Engineering of Programming* 1.2, 18:1–18:23. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2017/1/18.

Myers, G. J., T. Badgett and C. Sandler, eds. (Jan. 2012). *The Art of Software Testing*. 1st ed. Wiley. ISBN: 978-1-119-20248-6. DOI: 10.1002/9781119202486.

Nayak, S., R. Agarwal and S. K. Khatri (Jan. 2022). "Automated Assessment Tools for Grading of Programming Assignments: A Review". In: *2022 International Conference on Computer Communication and Informatics (ICCCI)*. Coimbatore, India: IEEE, pp. 1–4. ISBN: 978-1-66548-035-2. DOI: 10.1109/ICCCI54379.2022.9740769.

Nguyen, V., S. Deeds-Rubin, T. Tan and B. Boehm (2007). "A SLOC Counting Standard". In: *The 22nd International Annual Forum on COCOMO II and Systems/SoftwareCost Modeling*. Vol. 2007. Los Angeles, USA, pp. 1–16.

Nurue, H. D. and J. Gray (Apr. 2024). “A Testing Extension for Scratch”. In: *Proceedings of the 2024 ACM Southeast Conference on ZZZ*. Marietta, USA: ACM, pp. 266–271. ISBN: 9798400702372. DOI: 10.1145/3603287.3651217.

Nystrom, R. (2014). *Game Programming Patterns*. Los Gatos: Genever Benning. ISBN: 978-0-9905829-1-5.

Obermüller, F., L. Bloch, L. Greifenstein, U. Heuer and G. Fraser (Oct. 2021). “Code Perfumes: Reporting Good Code to Encourage Learners”. In: *Proceedings of the 16th Workshop in Primary and Secondary Computing Education*. Germany (virtual): ACM, pp. 1–10. ISBN: 978-1-4503-8571-8. DOI: 10.1145/3481312.3481346.

Oliveira, E. C., R. A. Bittencourt and R. P. Trindade (Oct. 2019). “Introduction to Computational Thinking for K-12 Educators through Distance Learning”. In: *2019 IEEE Frontiers in Education Conference (FIE)*. Covington, USA: IEEE, pp. 1–9. ISBN: 978-1-72811-746-1. DOI: 10.1109/FIE43999.2019.9028492.

Orrell, J. (Oct. 2006). “Feedback on Learning Achievement: Rhetoric and Reality”. In: *Teaching in Higher Education* 11.4, pp. 441–456. ISSN: 1356-2517, 1470-1294. DOI: 10.1080/13562510600874235.

Ota, G., Y. Morimoto and H. Kato (Sept. 2016). “Ninja Code Village for Scratch: Function Samples/Function Analyser and Automatic Assessment of Computational Thinking Concepts”. In: *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Cambridge, UK: IEEE, pp. 238–239. ISBN: 978-1-5090-0252-8. DOI: 10.1109/VLHCC.2016.7739695.

Paiva, J. C., J. P. Leal and Á. Figueira (Sept. 2022). “Automated Assessment in Computer Science Education: A State-of-the-Art Review”. In: *ACM Transactions on Computing Education* 22.3, pp. 1–40. ISSN: 1946-6226, 1946-6226. DOI: 10.1145/3513140.

Paiva, J. C., R. Queirós, J. P. Leal and J. Swacha (2020). “Yet Another Programming Exercises Interoperability Language”. In: 9th Symposium on Languages, Applications and Technologies. OpenAccess Series in Informatics (OASIcs). Portugal (virtual): Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 14:1–14:8. DOI: 10.4230/OASIcs.SLATE.2020.14.

Pan, J. (1999). *Software Testing*. Dependable Embedded Systems. URL: https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/.

Pasternak, E., R. Fenichel and A. N. Marshall (Oct. 2017). “Tips for Creating a Block Language with Blockly”. In: *2017 IEEE Blocks and Beyond Workshop (B&B)*. Raleigh, USA: IEEE, pp. 21–24. ISBN: 978-1-5386-2480-7. DOI: 10.1109/BLOCKS.2017.8120404.

Petit, J., S. Roura, J. Carmona, J. Cortadella, J. Duch, O. Gimnez, A. Mani, J. Mas, E. Rodrguez-Carbonell, E. Rubio, E. d. S. Pedro and D. Venkataramani (July 2018). “Jutge.Org: Characteristics and Experiences”. In: *IEEE Transactions on Learning Technologies* 11.3, pp. 321–333. ISSN: 1939-1382. DOI: 10.1109/TLT.2017.2723389.

Peveler, M., E. Maicus and B. Cutler (Feb. 2019). “Comparing Jailed Sandboxes vs Containers within an Autograding System”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. Minneapolis, USA: ACM, pp. 139–145. ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287507.

Pieterse, V. (Apr. 2013). “Automated Assessment of Programming Assignments”. In: *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*. Arnhem, Nederland: Open Universiteit Heerlen, pp. 45–56. URL: https://dl.acm.org/doi/10.5555/2541917.2541921.

Pirttinen, N., V. Kangas, I. Nikkarinen, H. Nygren, J. Leinonen and A. Hellas (July 2018). "Crowdsourcing Programming Assignments with CrowdSorcerer". In: *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. Larnaca, Cyprus: ACM, pp. 326–331. ISBN: 978-1-4503-5707-4. DOI: 10.1145/3197091.3197117.

Queirós, R. and J. P. Leal (June 2011). "Pexil: Programming Exercises Interoperability Language". In: *Conferência Nacional XATA: XML, Aplicações e Tecnologias Associadas, 9*. Vila do Conde, Portugal: ESEIG, pp. 37–48. ISBN: 978-989-96863-1-1. URL: http://hdl.handle.net/10400.22/4748.

Queirós, R. and J. P. Leal (2012). "Programming Exercises Evaluation Systems". In: *Proceedings of the 4th International Conference on Computer Supported Education*. Vol. 2. Porto, Portugal: SciTePress, pp. 83–90. ISBN: 978-989-8565-06-8. DOI: 10.5220/0003924900830090.

Queirós, R. and J. P. Leal (Jan. 2013). "BabeLO—An Extensible Converter of Programming Exercises Formats". In: *IEEE Transactions on Learning Technologies* 6.1, pp. 38–45. ISSN: 1939-1382. DOI: 10.1109/TLT.2012.21.

Resnick, M., J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman and Y. Kafai (Nov. 2009). "Scratch: Programming for All". In: *Communications of the ACM* 52.11, pp. 60–67. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1592761.1592779.

Resnick, M. and N. Rusk (Oct. 2020). "Coding at a Crossroads". In: *Communications of the ACM* 63.11, pp. 120–127. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3375546.

Revilla, M. A., S. Manzoor and R. Liu (2008). "Competitive Learning in Informatics: The UVa Online Judge Experience". In: *Olympiads in Informatics* 2.10, pp. 131–148. URL: https://ioinformatics.org/journal/INFOL035.pdf.

Robins, A., J. Rountree and N. Rountree (June 2003). "Learning and Teaching Programming: A Review and Discussion". In: *Computer Science Education* 13.2, pp. 137–172. ISSN: 0899-3408, 1744-5175. DOI: 10.1076/csed.13.2.137.14200.

Romli, R., S. Sulaiman and K. Z. Zamli (June 2010). "Automatic Programming Assessment and Test Data Generation a Review on Its Approaches". In: *2010 International Symposium on Information Technology*. Kuala Lumpur, Malaysia: IEEE, pp. 1186–1192. ISBN: 978-1-4244-6715-0. DOI: 10.1109/ITSIM.2010.5561488.

Ronacher, A. and D. Lord (Mar. 2022). *Jinja2*. Version 3.1. Pallets Projects. URL: https://jinja.palletsprojects.com/en/3.1.x/.

Rosenberg, J. B. (Oct. 1996). *How Debuggers Work: Algorithms, Data Structures, and Architecture*. 1st ed. Wiley. 272 pp. ISBN: 978-0-471-14966-8.

Runeson, P. (July 2006). "A Survey of Unit Testing Practices". In: *IEEE Software* 23.4, pp. 22–29. ISSN: 0740-7459. DOI: 10.1109/MS.2006.91.

Sarsa, S., J. Leinonen, C. Koutcheme and A. Hellas (Sept. 2022). "Speeding up Automated Assessment of Programming Exercises". In: *Proceedings of the 2022 Conference on United Kingdom & Ireland Computing Education Research*. Dublin, Ireland: ACM, pp. 1–7. ISBN: 978-1-4503-9742-1. DOI: 10.1145/3555009.3555013.

Savidis, A. and C. Savaki (2020). "Complete Block-Level Visual Debugger for Blockly". In: *Proceedings of the 2nd International Conference on Human Systems Engineering and Design*. Ed. by T. Ahram, W. Karwowski, S. Pickl and R. Taiar. Advances in Intelligent Systems and Computing. Munich, Germany: Springer, pp. 286–292. ISBN: 978-3-030-27928-8. DOI: 10.1007/978-3-030-27928-8_43.

Sax, L. J., K. J. Lehman and C. Zavala (Mar. 2017). “Examining the Enrollment Growth: Non-CS Majors in CS1 Courses”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. Seattle, USA: ACM, pp. 513–518. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017781.

Schlueter, I. Z., M. Layman, L. Timmermans, B. P. Kinoshita and C. Granum (2022). *TAP14 - The Test Anything Protocol V14*. Version v14. URL: https://testanything.org/tap-version-14-specification.html.

Scratch Addons (2023). URL: https://scratchaddons.com/.

Scratch Foundation (2022). *Growing a Global Creative Learning Movement: Scratch Foundation 2022 Annual Report*. Scratch Foundation. URL: https://www.scratchfoundation.org/annualreport.

Sels, B., P. Dawyndt, B. Mesuere, N. Strijbol and C. Van Petegem (2021). “TESTed: programmeertaal-onafhankelijk testen van oplossingen voor programmeeroefeningen : Eenvoudig oefeningen opstellen met een DSL”. MA thesis. Universiteit Gent. URL: http://lib.ugent.be/catalog/rug01:003008250.

Shadish, W. R., T. D. Cook and D. T. Campbell (2002). *Experimental and Quasi-experimental Designs for Generalized Causal Inference*. 2nd ed. Experimental and Quasi-experimental Designs for Generalized Causal Inference v. 1. Houghton Mifflin. ISBN: 978-0-395-61556-0.

Shen, V. Y., S. D. Conte and H. E. Dunsmore (Mar. 1983). “Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support”. In: *IEEE Transactions on Software Engineering* SE-9.2, pp. 155–165. ISSN: 0098-5589. DOI: 10.1109/TSE.1983.236460.

Shore, J. (Sept. 2004). “Fail Fast [Software Debugging]”. In: *IEEE Software* 21.5, pp. 21–25. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2004.1331296.

Shute, V. J. (Mar. 2008). “Focus on Formative Feedback”. In: *Review of Educational Research* 78.1, pp. 153–189. ISSN: 0034-6543. DOI: 10.3102/0034654307313795.

Simões, A. and R. Queirós (2020). “On the Nature of Programming Exercises”. In: *First International Computer Programming Education Conference (ICPEC 2020)*. Open Access Series in Informatics (OASIcs). Vila do Conde, Portugal: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:9. DOI: 10.4230/OASICS.ICPEC.2020.24.

Simon (2015). “Emergence of Computing Education as a Research Discipline”. PhD thesis. Helsinki, Finland: Alto University. 100 pp. URL: http://urn.fi/URN:ISBN:978-952-60-6416-1.

Stahlbauer, A., C. Frädrich and G. Fraser (Dec. 2020). “Verified from Scratch: Program Analysis for Learners’ Programs”. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. Australia (virtual): ACM, pp. 150–162. ISBN: 978-1-4503-6768-4. DOI: 10.1145/3324884.3416554.

Stahlbauer, A., M. Kreis and G. 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*. Tallin, Estonia: ACM, pp. 165–175. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338910.

Staubitz, T., H. Klement, J. Renz, R. Teusner and C. Meinel (Dec. 2015). “Towards Practical Programming Exercises and Automated Assessment in Massive Open Online Courses”. In: *2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. Zhuhai, China: IEEE, pp. 23–30. ISBN: 978-1-4673-9226-6. DOI: 10.1109/TALE.2015.7386010.

Staubitz, T., R. Teusner and C. Meinel (Dec. 2017). "Towards a Repository for Open Auto-Gradable Programming Exercises". In: *2017 IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. Hong Kong, China: IEEE, pp. 66–73. ISBN: 978-1-5386-0900-2. DOI: 10.1109/TALE.2017.8252306.

Strickroth, S., M. Striewe, O. Müller, U. Priss, S. Becker, O. Rod, R. Garmann, O. J. Bott and N. Pinkwart (2015). "ProFormA: An XML-based Exchange Format for Programming Tasks". In: *eleed* 11.1. ISSN: 1860-7470. URL: http://nbn-resolving.de/urn:nbn:de:0009-5-41389.

Striewe, M. (Nov. 2016). "An Architecture for Modular Grading and Feedback Generation for Complex Exercises". In: *Science of Computer Programming* 129, pp. 35–47. ISSN: 01676423. DOI: 10.1016/j.scico.2016.02.009.

Strijbol, N., P. Dawyndt, B. Mesuere and C. Van Petegem (2020). "TESTed: One Judge to Rule Them All". MA thesis. Universiteit Gent. URL: http://lib.ugent.be/catalog/rug01:002836313.

Strijbol, N., R. De Proft, K. Goethals, B. Mesuere, P. Dawyndt and C. Scholliers (Feb. 2024). "Blink: An Educational Software Debugger for Scratch". In: *SoftwareX* 25, p. 101617. ISSN: 23527110. DOI: 10.1016/j.softx.2023.101617.

Strijbol, N., C. Scholliers and P. Dawyndt (June 2023). "Blink: An Educational Software Debugger for Scratch". In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education*. Vol. 2. Turku, Finland: ACM, pp. 648–648. ISBN: 9798400701399. DOI: 10.1145/3587103.3594189.

Strijbol, N., B. Sels, C. Van Petegem, R. Maertens, C. Scholliers, B. Mesuere and P. Dawyndt (2024). "TESTed-DSL: A Domain-Specific Language to Create Programming Exercises with Language-Agnostic Automated Assessment". In: *Software Testing, Verification & Reliability*. Manuscript submitted for publication.

Strijbol, N., C. Van Petegem, R. Maertens, B. Sels, C. Scholliers, P. Dawyndt and B. Mesuere (May 2023). "TESTed: An Educational Testing Framework with Language-Agnostic Test Suites for Programming Exercises". In: *SoftwareX* 22, p. 101404. ISSN: 2352-7110. DOI: 10.1016/j.softx.2023.101404.

Sundarram, K. (July 2022). *Please Stop Citing TIOBE*. Krishna's personal blog. URL: https://blog.nindalf.com/posts/stop-citing-tiobe/.

Sutherland, A. V. and A. Booker (Sept. 2019). "Sums of Three Cubes". Computational Mathematics Colloquium (Waterloo). URL: https://math.mit.edu/~drew/Waterloo2019.pdf.

Swacha, J. (2018). "SIPE: A Domain-Specific Language for Specifying Interactive Programming Exercises". In: *Towards a Synergistic Combination of Research and Practice in Software Engineering*. Ed. by P. Kosiuczenko and L. Madeyski. Vol. 733. Springer, pp. 15–29. ISBN: 978-3-319-65208-5. DOI: 10.1007/978-3-319-65208-5_2.

Tang, T., R. Smith, S. Rixner and J. Warren (July 2016). "Data-Driven Test Case Generation for Automated Programming Assessment". In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. Arequipa, Peru: ACM, pp. 260–265. ISBN: 978-1-4503-4231-5. DOI: 10.1145/2899415.2899423.

Techapalokul, P. and E. Tilevich (Oct. 2017). "Quality Hound — An Online Code Smell Analyzer for Scratch Programs". In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Raleigh, USA: IEEE, pp. 337–338. ISBN: 978-1-5386-0443-4. DOI: 10.1109/VLHCC.2017.8103498.

Tedre, M., Simon and L. Malmi (Apr. 2018). “Changing Aims of Computing Education: A Historical Survey”. In: *Computer Science Education* 28.2, pp. 158–186. ISSN: 0899-3408, 1744-5175. DOI: 10.1080/08993408.2018.1486624.

Timmis, S., P. Broadfoot, R. Sutherland and A. Oldfield (2016). “Rethinking Assessment in a Digital Age: Opportunities, Challenges and Risks”. In: *British Educational Research Journal* 42.3, pp. 454–476. ISSN: 1469-3518. DOI: 10.1002/berj.3215.

TIOBE (May 2024). *TIOBE Index for May 2024*. TIOBE Software. URL: https://web.archive.org/web/20240531151836/https://www.tiobe.com/tiobe-index/.

Truong, N., P. Bancroft and P. Roe (Sept. 2005). “Learning to Program through the Web”. In: *ACM SIGCSE Bulletin* 37.3, pp. 9–13. ISSN: 0097-8418. DOI: 10.1145/1151954.1067452.

Ullah, Z., A. Lajis, M. Jamjoom, A. Altalhi, A. Al-Ghamdi and F. Saleem (2018). “The Effect of Automatic Assessment on Novice Programming: Strengths and Limitations of Existing Systems”. In: *Computer Applications in Engineering Education* 26.6, pp. 2328–2341. ISSN: 1099-0542. DOI: 10.1002/cae.21974.

Ungar, D., H. Lieberman and C. Fry (Apr. 1997). “Debugging and the Experience of Immediacy”. In: *Communications of The ACM* 40.4, pp. 38–43. ISSN: 0001-0782. DOI: 10.1145/248448.248457.

Valente, L., A. Conci and B. Feijó (Nov. 2005). “Real Time Game Loop Models for Single-Player Computer Games”. In: *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*. Vol. 89. São Paulo, Brazil, p. 99.

Van Petegem, C., L. Deconinck, D. Mourisse, R. Maertens, N. Strijbol, B. Dhoedt, B. De Wever, P. Dawyndt and B. Mesuere (Mar. 2023). “Pass/Fail Prediction in Programming Courses”. In: *Journal of Educational Computing Research* 61.1, pp. 68–95. ISSN: 0735-6331, 1541-4140. DOI: 10.1177/07356331221085595.

Van Petegem, C., K. Demeyere, R. Maertens, N. Strijbol, B. De Wever, B. Mesuere and P. Dawyndt (Apr. 2024). *Mining Patterns in Syntax Trees to Automate Code Reviews of Student Solutions for Programming Exercises*. Manuscript submitted for publication. arXiv: 2405.01579. Pre-published.

Van Petegem, C., R. Maertens, N. Strijbol, J. Van Renterghem, F. Van Der Jeugt, B. De Wever, P. Dawyndt and B. Mesuere (Dec. 2023). “Dodona: Learn to Code with a Virtual Co-Teacher That Supports Active Learning”. In: *SoftwareX* 24, p. 101578. ISSN: 23527110. DOI: 10.1016/j.softx.2023.101578.

Verhoeff, T. (2008). “Programming Task Packages: Peach Exchange”. In: *Olympiads in Informatics* 8, pp. 192–207. URL: https://ioinformatics.org/journal/INFOL019.pdf.

Voeten, I., P. Dawyndt, C. Scholliers and N. Strijbol (2023). “Een blokgebaseerd testframework voor Scratch.” MA thesis. Universiteit Gent. URL: http://lib.ugent.be/catalog/rug01:003150096.

Wang, B. L. and E. Klopfer (2021). “Developing Resources for Debugging Education Using Block-Based Languages”. MA thesis. Massachusetts Institute of Technology. URL: https://hdl.handle.net/1721.1/139091.

Wang, W., C. Zhang, A. Stahlbauer, G. Fraser and T. Price (June 2021). “SnapCheck: Automated Testing for Snap *!* Programs”. In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education*. Germany (virtual): ACM, pp. 227–233. ISBN: 978-1-4503-8214-4. DOI: 10.1145/3430665.3456367.

Wangenheim, C. G. V., J. C. R. Hauck, M. F. Demetrio, R. Pelle, N. D. Cruz Alves, H. Barbosa and L. F. Azevedo (Apr. 2018). "CodeMaster - Automatic Assessment and Grading of App Inventor and Snap! Programs". In: *Informatics in Education* 17.1, pp. 117–150. ISSN: 1648-5831, 2335-8971. DOI: 10.15388/infedu.2018.08.

Wasik, S., M. Antczak, J. Badura, A. Laskowski and T. Sternal (Jan. 2018). "A Survey on Online Judge Systems and Their Applications". In: *ACM Computing Surveys* 51.1, pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/3143560.

Weintrop, D. and U. Wilensky (June 2015). "To Block or Not to Block, That Is the Question: Students' Perceptions of Blocks-Based Programming". In: *Proceedings of the 14th International Conference on Interaction Design and Children*. Boston, USA: ACM, pp. 199–208. ISBN: 978-1-4503-3590-4. DOI: 10.1145/2771839.2771860.

Wilcox, C. (Feb. 2016). "Testing Strategies for the Automated Grading of Student Programs". In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. Memphis, USA: ACM, pp. 437–442. ISBN: 978-1-4503-3685-7. DOI: 10.1145/2839509.2844616.

Wilkinson, M. D. et al. (Mar. 2016). "The FAIR Guiding Principles for Scientific Data Management and Stewardship". In: *Scientific Data* 3.1 (1), p. 160018. ISSN: 2052-4463. DOI: 10.1038/sdata.2016.18.

Winters, T., T. Manshreck and H. Wright (Feb. 2020). *Software Engineering at Google: Lessons Learned from Programming over Time*. O'Reilly Media. 602 pp. ISBN: 978-1-4920-8276-7. Google Books: V3TTDwAAQBAJ. URL: https://www.oreilly.com/library/view/software-engineering-at/9781492082781/.

Zavala, L. and B. Mendoza (Feb. 2018). "On the Use of Semantic-Based AIG to Automatically Generate Programming Exercises". In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. Baltimore, USA: ACM, pp. 14–19. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159608.

Zeller, A. (June 2009). *Why Programs Fail: A Guide to Systematic Debugging*. 2nd ed. San Francisco, USA: Morgan Kaufmann. 544 pp. ISBN: 978-0-12-374515-6. URL: https://www.sciencedirect.com/book/9780123745156/why-programs-fail.

Zhang, L. and J. Nouri (Nov. 2019). "A Systematic Review of Learning Computational Thinking through Scratch in K-9". In: *Computers & Education* 141, p. 103607. ISSN: 03601315. DOI: 10.1016/j.compedu.2019.103607.

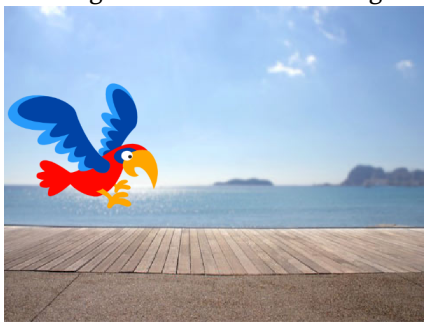
Zinovieva, I. S., V. O. Artemchuk, A. V. Iatsyshyn, O. O. Popov, V. O. Kovach, A. V. Iatsyshyn, Y. O. Romanenko and O. V. Radchenko (Mar. 2021). "The Use of Online Coding Platforms as Additional Distance Tools in Programming Education". In: *XII International Conference on Mathematics, Science and Technology Education*. Vol. 1840. Journal of Physics: Conference Series. Kryvy Rih, Ukraine: IOP, p. 012029. DOI: 10.1088/1742-6596/1840/1/012029.

Appendix A.

Task description of the VPW

Below is a translated version of the second Scratch exercise from the 2017 edition of the Flemish Programming Contest (Vlaamse Programmeerwedstrijd, VPW*).*

Problem 02 (10 points)
The stage looks like the drawing below.



Write a program that lets the parrot fly from left to right. When he touches the edge, he must turn around. The program may only start when the parrot is clicked. (Tip: use the "change costume to ..." block for flying)

Below is a possible solution: