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.

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

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.

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 that help 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 considered 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.