Abstractions for Third-Party Extensibility of Educational Programming Environments

by

Trevor Paley

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2023

APPROVED:

Professor Charles Roberts, Thesis Advisor

Professor George Heineman, Reader

Professor Craig Shue, Head of Department

Abstract

While many companies and organizations have built applications which use collaborative features to help teach computer science in classrooms, they have traditionally provided very limited support for third-party developers to build on top of their platforms to add new features and interfaces. Using our own educational programming environment Necode, we develop a set of APIs and abstractions to enable third-party developers to integrate programming languages, design in-class activities, and implement algorithms (policies) for automatically generating peer-to-peer networks for real-time communication between students.

To prove the viability of our APIs, we translated all of Necode's built-in modules (consisting of languages, activities, and policies) to use these APIs, as well as developing new brand modules on top of them. We then demonstrated that these modules could be statically unlinked from the Necode build and packaged into a plugin for dynamic hot installation without requiring a re-compile of Necode or any downtime.

Table of Contents

List of Figures

Chapter 1

Introduction

Computer science education has attracted attention from software developers since its conception, which makes sense given that computer science educators often *are* software developers themselves, and are perfectly capable of building software to support their classroom. However, not every instructor has the time or desire to build their own education software from scratch, and so educators [1, 2], organizations [3], and for-profit companies [4, 5] have created more generalized educational programming environments that instructors can drop into their classrooms and configure as needed. But with pre-built solutions comes a cost: instructors get what the developers give them. No longer can a computer science teacher alter the environment to fit their precise specifications, instead they need to alter their specifications to fit the environment. Sometimes, an environment will be open-source, allowing instructors to change it as they need—in theory, at least. Navigating large projects can be hard, and even if an instructor is able to make their desired changes, it may be of little help if the critically important online interactivity features will only work with the official build, like in Scratch [6].

The solution is for educational programming environments to provide support for third-party extensibility so that third-parties (e.g. instructors) can alter aspects of the environment in a controlled manner without needing to start from the ground up. Although third-party extensions are a well-known mechanism for enabling innovation, especially in computer science [7], relatively few educational programming environments support any kind of third-party extensibility. And even those that do such as Scratch and DrRacket [8] only allow extensions insofar as they add on top of a strongly established design, rather than allowing extensions to alter things in any meaningful way.

Last year, we created Necode, a brand new educational programming environment aimed at supporting in-class activities, but also aimed at having a modular design which could theoretically support third-party plugins in the future [9]. In this thesis, we continue that work, delivering a large set of APIs and abstractions for third-party developers to implement plugins for Necode. These plugins are hot-loadable and can include anything from new activities/user experiences to new language support to new ways of connecting users together and building social and collaborative real-time interaction. And importantly, we demonstrate that these APIs are powerful enough for developers to build non-trivial extensions and run significantly different kinds of activities than what we had created and described for our original report. Beyond Necode, our hope is that by creating these abstractions and thoroughly describing both their design and the rationale for their design, this thesis can provide a foundation for other educational programming environments to build out effective third-party extensibility APIs as well.

Chapter 2

Background

In this chapter, we look at some of the prior work in both educational programming environments in general, and in their extensibility. We start by surveying a few educational programming environments currently in use. Then we look at a couple of past approaches to extensibility in educational programming environments. Finally we give a brief review of Necode, our own educational programming environment that we design and implement our new abstractions on top of.

2.1 Features of Educational Programming Environments

Educational programming environments are highly varied (possibly more so than "professional" IDEs) due to a wide array of pedagogical concerns and beliefs. While we ultimately implement our abstractions and developer APIs on top of Necode, we still need to understand what kinds of features other educational programming environments provide in order to inform what our new APIs and abstractions should allow developers to do.

To guide this search, we selected a small list of applications which we can use to get a range of how different features are used in different environments. These specific applications were selected to provide a sample of environments that are actually used at WPI, environments with significant corporate backing, and environments we encountered in our research which offer social/collaborative features. These are:

- Scratch [3]
- p5.js Web Editor [10]
- CodeCircle [1]

- Replit Teams for Education [4]
- Coding Rooms [5]
- DrRacket [11]

A summarized feature comparison between these environments is available in Figure 2.1. These do not comprise a comprehensive set of features in educational programming environments, but they do help provide a high-level summary of what different environments can do.

	Scratch	p5.js	CodeCircle	Replit	Coding Rooms	DrRacket
Visual Programming	✓	✗	✗	✓*	✗	✓*
Poly-lingual	✗	✗	✗	✓	✓	✓†
REPL	✓‡	✗	✗	✓	✓	✓
Shared editors	✗	✗	✓	✓	✓	✗
Remixing	✓	✓	✓	✓	✓	✗
Graphical Output	✓	✓	✓	✓	✓	✓
Auto-reload	✗	✓	✓	✗	✗	✗
Classrooms	✓	✗	✗	✓	✓	✗
Auto-grading	✗	✗	✗	✓	✓	✗
Price	Free	Free	Free	Free	Undisclosed	Free
Open-source	✓	✓	✗	✗	✗	✓

* In some cases supports images being interspersed within text-based code
† Typically only Racket/Lisp-like languages are used
‡ Clicking on blocks to run them is effectively a REPL

Figure 2.1: *Feature comparison of a few educational programming environments*

2.1.1 Visual Programming

While text-based programming is very common, some educational programming environments like Scratch have special code editors designed around visual or "block-based" programming. The idea of such languages is to be a softer introduction to programming which removes the difficulties of learning syntax by eliminating the possibility of syntax errors altogether.

Partially to reduce conceptual load for beginners and partially because visual programming editors are less generalizable than text editors, educational programming environments based on a visual programming language typically only have a single language and are laser-focused on optimizing the experience for that language. Thus, it is unlikely that Necode would be able to offer a generalized version of a language like Scratch which competes with the first-party application, but theoretical support for visual programming is still worth considering as future work.

Some text-based programming environments also allow the inclusion of some visual elements. For example, some of DrRacket's [11] and Replit's [12] supported languages allow images to be placed into otherwise text-based code.

2.1.2 Poly-Lingual

Unlike most of the other environments we are looking at, Replit, Coding Rooms, and DrRacket are not limited to one or a small subset of languages. Rather than running user code on a user's client, Replit and Coding Rooms run user code on virtual machines in the cloud, allowing code in any language (at least, any language with Linux support) to be executed. This results in a user experience which is abstracted for any language rather than the more targeted experience with one-language environments like Scratch, p5.js, and CodeCircle.

DrRacket has a somewhat different approach to multiple-language support in which other languages can be implemented in Racket and then added to DrRacket in a plugin (see Section 2.2.2 for more discussion of DrRacket's plugin system). However, while completely unrelated languages like C and Python *can* be used in DrRacket, it is primarily focused on Racket- and Lisp-like languages, and has a user experience specifically tailored to that.

To balance user experience with development time, both generalized and tailored experiences are valuable. Providing a way for third-party developers to do both is thus a goal of this thesis.

2.1.3 Read-Eval-Print Loops

Being able to evaluate small chunks of code at will can be an exceptionally helpful educational tool for students. One of the principals in developing DrScheme's (the precursor to DrRacket) REPL interface was that state should not be preserved between executions [2]. That is, any mutation to program state that is performed in the REPL should not persist when the program is re-run. These days that design is the standard for read-eval-print loops, but we can see in Scratch that sometimes even with persistent state, being able to run code on the fly to see what it does can be helpful.

Educational programming environments like Replit and Coding Rooms are less targeted than DrRacket or Scratch and offer an abstracted terminal interface for any language which has a REPL implementation. This model will not be able to provide a specialized UX features like DrRacket and Scratch offer for their respective first-class language(s), but it is often sufficient, and is a good starting place for generalizing REPLs in Necode.

2.1.4 Shared Editors

One of the benefits of an educational programming environment being a web application is the opportunities that the internet provides to add collaboration features. A very popular collaboration feature, one shared by CodeCircle, Replit, and Coding Rooms, is the shared editor, in which multiple users can write code in the same editor at the same time in a manner similar to Google Docs. This allows students to directly collaborate on code rather than conforming strictly to peer programming (although it can help even in peer programming scenarios if the programmers change roles frequently). Unlike some of the other features, the shared editor experience is largely unvaried and seems like it could be abstracted to support nearly any scenario, at least as long as the code is predominantly text-based (e.g. Scratch may not be as conducive to a generalized approach).

Synchronized text editing is not a new problem, and solutions have largely coalesced around two main approaches: operational transformation (OT) [13] and conflict-free replicated data types (CRDT) [14]. Fortunately, JavaScript implementations of both techniques already exist in the ShareJS [15] and Yjs [16] libraries respectively. Both OT and CRDT work well and both are used for shared documents in the real world, though in peer-to-peer scenarios (like with Necode's RTC), CRDT tends to behave more favorably [17].

2.1.5 Remixing

Shared editors are certainly not the only form of collaboration. Every environment on our list (except DrRacket, which is not a web application) natively supports some form of creating a "remix" or "fork" of someone else's code. Scratch, CodeCircle, and Replit open this to the entire internet, where anyone can search a public list of projects others have made and create their own copy of a project by clicking a button. This is an asynchronous form of collaboration where different users can progressive iterate on each other's work.

Unlike most of the features here, asynchronous remixing is not a goal of Necode, since Necode is explicitly aimed at *in-class* activities, which are inherently constrained in time. With that said, allowing developers to implement synchronous code (and asset) sharing between users *is* a goal, beyond simply enabling shared editors.

2.1.6 Graphical Output

It is no secret that graphical output can be a more engaging teaching tool than plain text, and every single educational programming environment in our list supports visual output for code that users write. In Scratch, CodeCircle, and p5.js, graphics are the primary form of output (with them being the *only* form of output in Scratch), whereas Replit and Coding Rooms support graphical output as an additional pane for project types that support it. DrRacket takes a different approach from the rest and mixes images directly into text-based output, when supported. Suffice to say, it is a necessity that Necode allows third-party developers to allow users to produce graphics with their code.

2.1.7 Auto-Reload

While they do not have any form of read-eval-print loop like the other environments, p5.js and CodeCircle try to improve interactivity by allowing users to make the graphical output immediately and automatically refresh whenever code changes are made. The authors of CodeCircle claim that this “live mode” (as they call it) increases interactivity by having a lower feedback delay over needing to perform some other gesture to run code (such as clicking a “run” button, like one has to do in Replit, Coding Rooms, and DrRacket when running code in a non-REPL mode) [18].

We would like to note that there is a potential security risk when combining an auto-reload feature with shared editors (Section 2.1.4) in which an adversary with access to a shared editor could execute code on a victim’s computer without the victim being able to vet the code before execution. With that said, the viability of such an attack, especially if the identities of everyone sharing the editor are known, is limited.

2.1.8 Open-Source

While an application being open-source is not typically thought of as a *functional* feature, for the purposes of third-party extensibility (the focus of this thesis), it can actually be quite important. It is certainly possible to design plugin APIs which do not require developers to be able to see implementation details—in fact such designs are preferred—but having the ability to see how the underlying application is built can give third-party developers deeper insight into how their plugins interact with the system as a whole. Plus, when bugs or confusing/imprecise documentation inevitably appear, it can allow third-party developers to more easily identify what is going on and report an issue if necessary. Scratch, p5.js,[1] and DrRacket are three examples of open-source educational programming environments, and Necode has been and will continue to be one as well.

2.2 Extensibility of Existing Systems

Two of the environments we looked at in the previous section, Scratch and DrRacket, support plugins for third-party developers to add their own additional features on top of what comes with the application out-of-the-box. Seeing as this thesis is about the extensibility of educational programming environments, the details of how these plugin systems work is quite important, both to see what approaches have been taken in the past and to see in what areas the new plugin system we make for Necode improves upon prior work.

[1]During the development of a p5.js-based activity for Necode, having access to the p5.js source code made it much easier to identify undocumented behavior that was causing a bug and implement a workaround.

2.2.1 Scratch

Scratch 3.0 Extensions allow third-party developers to implement new blocks in JavaScript which users can include in their programs to do things that would otherwise be impossible or harder to achieve with Scratch's built-in blocks [6]. Further, this abstraction has been retroactively applied to core blocks such that all built-in blocks are just built-in extensions. This ensures that the extension API will always be at least powerful enough to implement built-ins, which while not *necessarily* sufficient, is a good sanity check to ensure that extensions pass a basic threshold of power.

While allowing third-party developers to implement new blocks greatly increases the potential capabilities of Scratch, blocks are the *only* thing that can be added with extensions. For example, a third-party developer could not add new options to sprites or implement a new menu for editing videos. The types of values that blocks can take in and return are also limited, so a third-party could not create a Scratch extension which provides Snap!'s first-class lists [19].

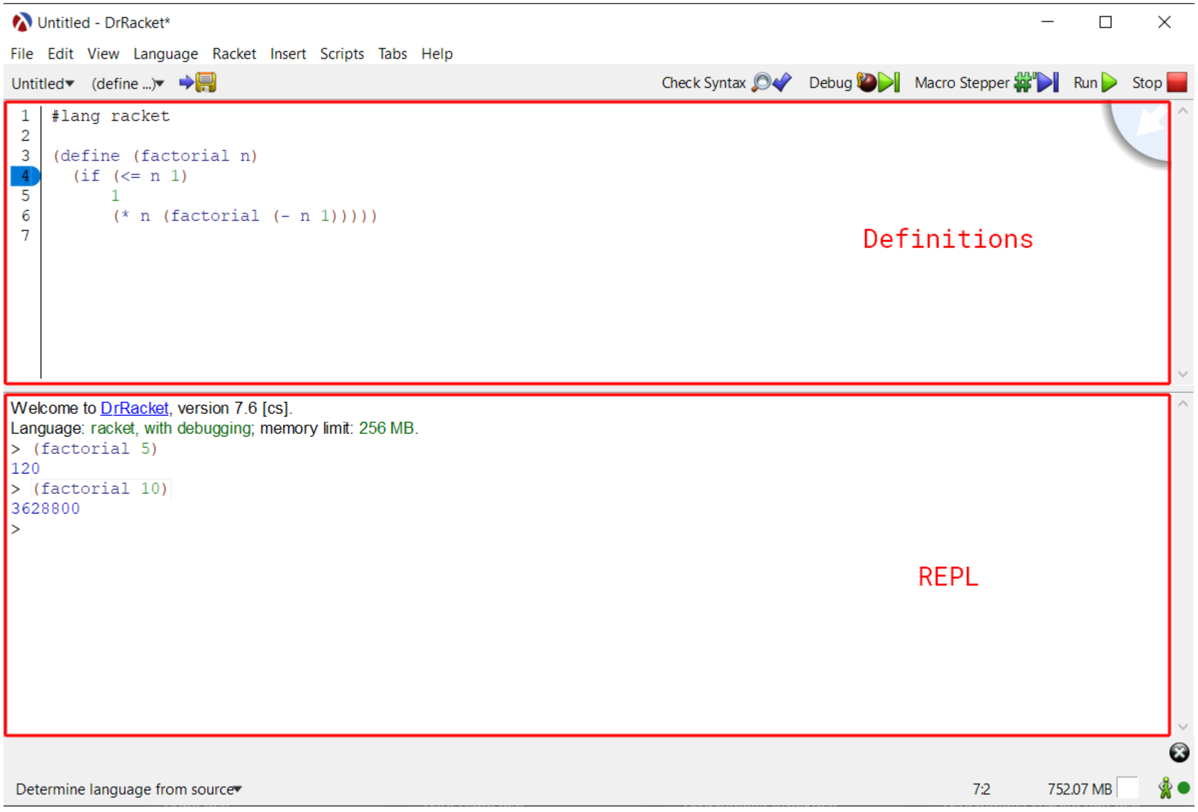
Additionally limiting Scratch's extensions is that while third parties may develop any extensions they want, those extensions cannot be used in the Scratch 3.0 editor on the Scratch website unless explicitly approved and added by the Scratch team. That makes sense given the considerable security risks of running arbitrary JavaScript, but it limits the usefulness of extensions for instructors. In order to use extensions, an instructor or institution must host their own instance of Scratch which will be lacking all of the web-based collaboration capabilities of the Scratch website.

2.2.2 DrRacket

DrRacket plugins are highly capable, being able to add new languages, integrate new languages with existing GUI, and add brand new GUI elements for languages to interact with [8]. This is done by writing plugins in Racket using DrRacket itself, leading to a scenario in which students themselves can create and use plugins, and instructors do not need separate development tools in order to add new features. This is an elegant approach, similar to how Emacs can be extended from its own self-hosted Emacs Lisp, though it is not clear if it has any practical benefits in a classroom setting where any plugins are likely to just be provided by the instructor anyways.

In order to let language definitions control how GUI elements should interact with them, Racket has a "capability" system in which developers can define capabilities that languages can provide values for and GUIs can access values of (for the current language). This allows for languages and GUIs to be implemented independently of each other which can improve modularity and compatibility of third-party plugins, while still allowing targeted interfaces to improve the experience of specific languages. Looking ahead, there are quite a few similarities between DrRacket's capabilities and the "feature" system that we develop for Necode (Section 3.2).

While DrRacket has utilities to add new GUI, its plugin API is not really designed to facilitate replacing the basic two-pane definitions/REPL GUI (Figure 2.2). In a similar vein, new languages must be able to be compiled to Racket. That makes things simple for languages built on top of Racket macros and it allows DrRacket to provide a debugging experience nearly for free, but it significantly complicates adding support for languages outside of the Lisp family. These limitations are not necessarily bad—they make sense for DrRacket's targeted design—but they do make DrRacket's plugin system less powerful than it otherwise could be.



```
#lang racket

(define (factorial n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

```
Welcome to DrRacket, version 7.6 [cs].
Language: racket, with debugging; memory limit: 256 MB.
> (factorial 5)
120
> (factorial 10)
3628800
>
```

Figure 2.2: *Screenshot of DrRacket's basic interface*

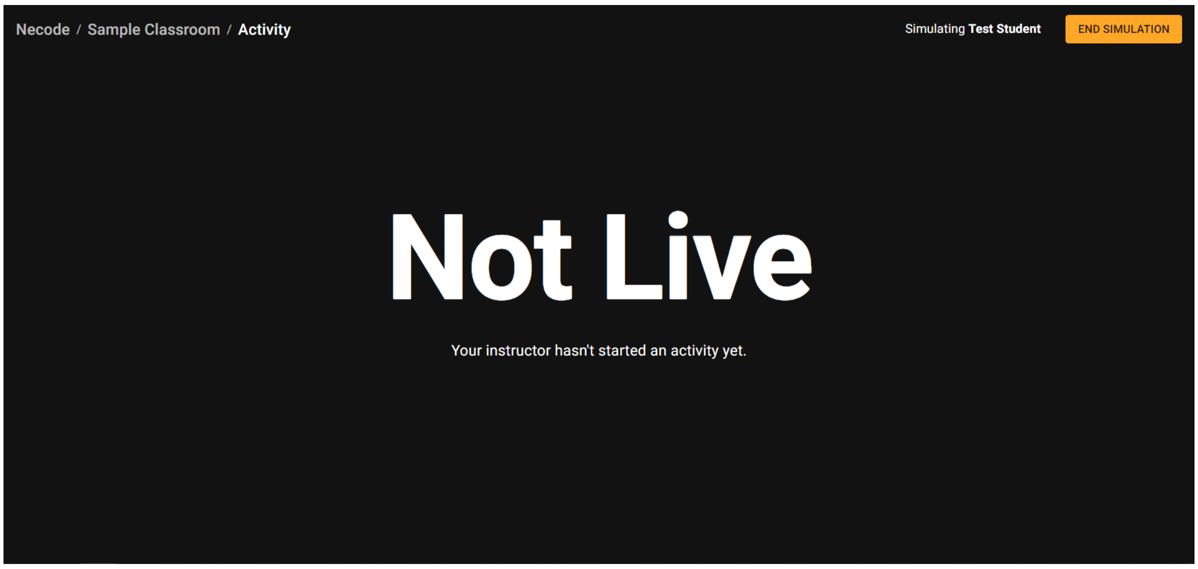
2.3 Necode

In 2021, we began work on Necode as a modular alternative to existing educational programming environments focused specifically on interactive in-class activities [9]. The internal design of Necode was relatively modular prior to this thesis, and that old design was iterated on to implement Necode's new third-party development APIs and abstractions. In this section, we will not discuss the entirety of Necode, nor will we dive into the details of Necode's original internal APIs, but we will provide sufficient background about how Necode works and how it was constructed prior to this thesis for readers to understand the rest of the research presented here.

2.3.1 The Student Experience

Like Coding Rooms, Replit Teams for Education, and in some cases Scratch, Necode organizes students into classrooms which are each managed by their respective instructors. An instructor can create a classroom, grab a "join code" (or join *link*) from the dashboard, and give it to students so they can join the class.

When a student joins a classroom, and every time they return to the classroom page, they will immediately be sent to the current in-class activity, or a waiting page if no activity is currently running (Figure 2.3).[2] Once an instructor starts an activity, the selected activity will automatically load on students' screens without any input from them. Figure 2.4 shows an example of what a student would see when an activity based on the "DOM Programming" activity type (Section 2.3.3.1) was started. Note that activities do not need to follow this UI—they can have any user interface that the activity developer wants to implement.



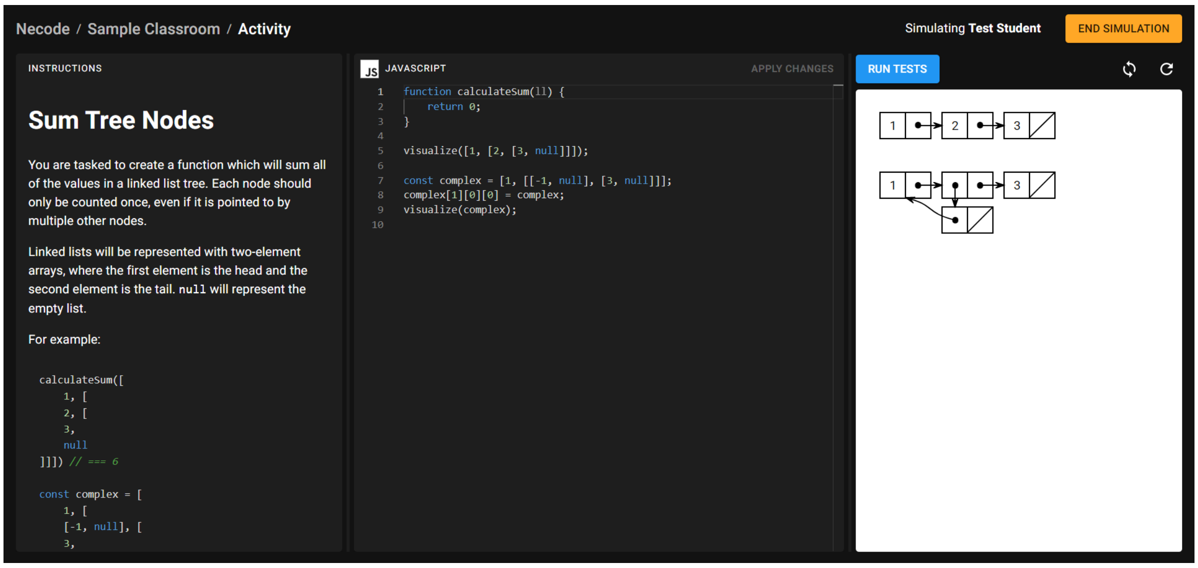
Not Live

Your instructor hasn't started an activity yet.

Figure 2.3: *The page students see when no activity is live*

Depending on which activity is live, students may be able to use the "submit" button to send what their code to the instructor to present to the whole class. Remember that Necode is focused on in-class activities, so these submissions are not like homework submissions and there is no mechanism for grading them—they are purely for instructors to view and present student work during the class period.

[2]The student-perspective screenshots are taken using a feature where instructors can simulate interacting with Necode as if they were a student. This feature is helpful for instructors when preparing their lessons and activities, though it is not particularly relevant to this thesis.



Sum Tree Nodes

You are tasked to create a function which will sum all of the values in a linked list tree. Each node should only be counted once, even if it is pointed to by multiple other nodes.

Linked lists will be represented with two-element arrays, where the first element is the head and the second element is the tail. `null` will represent the empty list.

For example:

```
calculateSum([
    1, [
    2, [
    3,
    null
]]]) // === 6

const complex = [
    1, [
    [-1, null], [
    3,
```

```javascript
function calculateSum(ll) {
    return 0;
}

visualize([1, [2, [3, null]]]);

const complex = [1, [[-1, null], [3, null]]];
complex[1][0][0] = complex;
visualize(complex);
```

Figure 2.4: *A sample activity from a student's perspective*

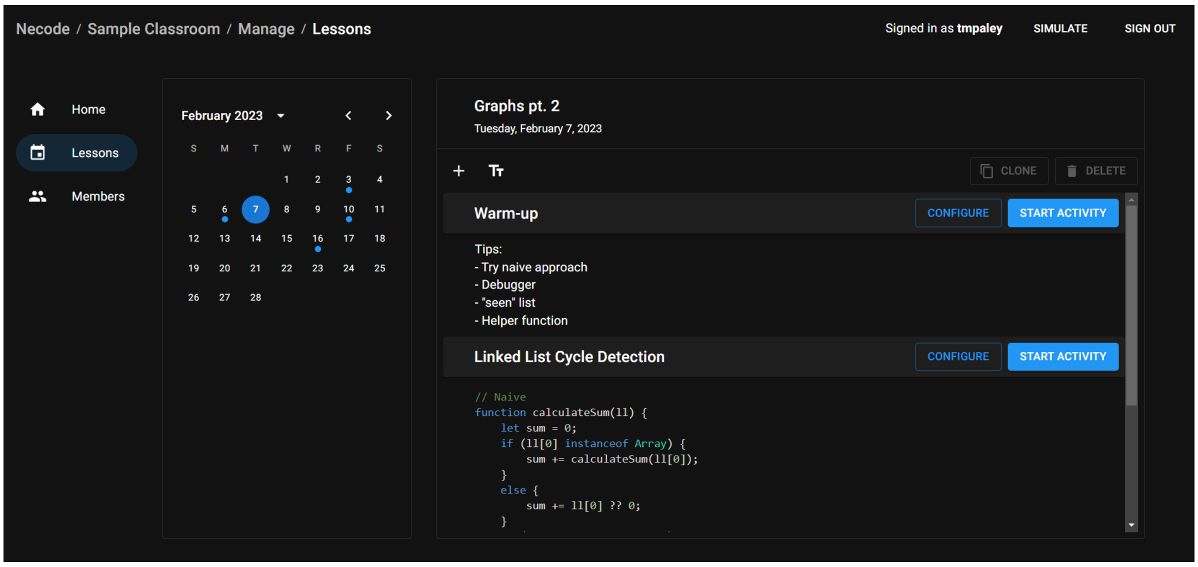
2.3.2 The Instructor Experience

When an instructor loads into a classroom they have made, they see a calendar where they can select dates and a panel where they can add activities to the lesson for that date using the toolbar (Figure 2.5). In the activities pane, each activity is displayed as a widget whose UI can be configured in the activity's code (e.g. the text notes in Figure 2.5 are a special type of "activity" which cannot actually be run solely consists of a highly customized widget). The instructor can then configure the activity by pressing the "configure" button, or run the activity by pressing "start activity." When they start an activity, they will see the same thing as the student would, but with a thin header which allows them to control the activity and view a list of student submissions.

When a configuration page is available (if it is not then the "configure" button does not appear on the widget), instructors will be sent to a new full-screen page to manage activity configuration options. These pages can be anything, though in the configuration page in Figure 2.6, it tries to closely mirror what the student sees. In order to test out the activity, the "preview" button can be pressed to load the activity with the current in-progress configuration. This will not cause the activity to start for students, but the previewed activity will also not have access to real-time communication features.

2.3.3 Activity Types

As we transition from the user's experience to the developer's, we should clarify some definitions. The term "activity" is somewhat imprecise because two things are referred to as activities. From an end-user experience, the actual instance that the instructor configures and runs is called



Graphs pt. 2
Tuesday, February 7, 2023

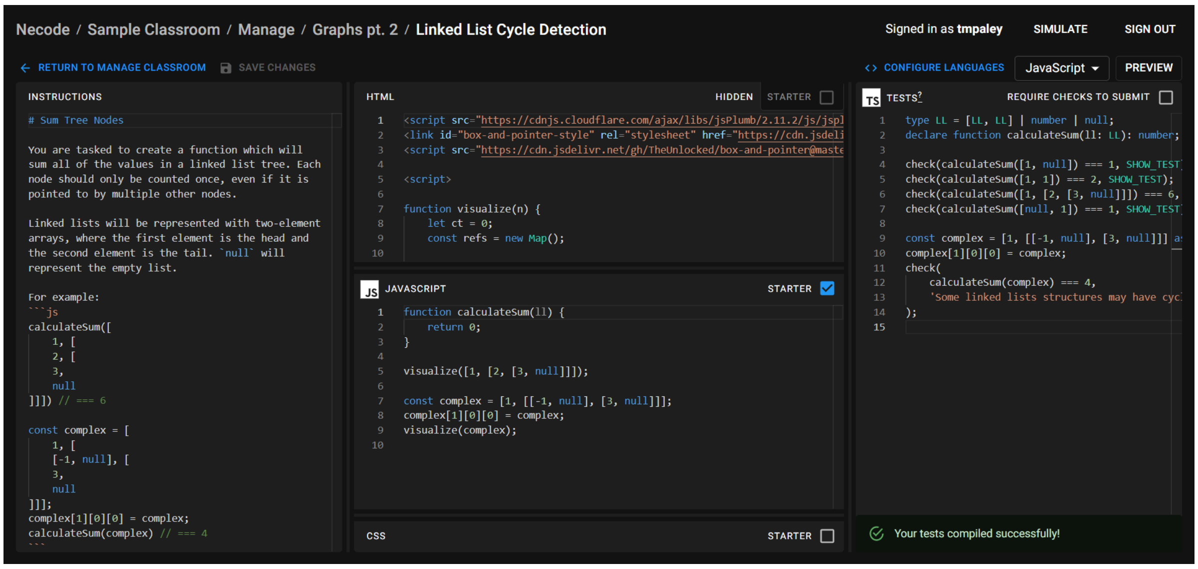
Warm-up

Tips:
- Try naive approach
- Debugger
- "seen" list
- Helper function

Linked List Cycle Detection

```js
// Naive
function calculateSum(ll) {
    let sum = 0;
    if (ll[0] instanceof Array) {
        sum += calculateSum(ll[0]);
    }
    else {
        sum += ll[0] ?? 0;
    }
```

Figure 2.5: *A sample lesson in the manage classroom page*



~~~
# Sum Tree Nodes

You are tasked to create a function which will
sum all of the values in a linked list tree. Each
node should only be counted once, even if it is
pointed to by multiple other nodes.

Linked lists will be represented with two-element
arrays, where the first element is the head and
the second element is the tail. `null` will
represent the empty list.

For example:
```js
calculateSum([
    1, [
    2, [
    3,
    null
]]]) // === 6

const complex = [
    1, [
    [-1, null], [
    3,
    null
]]];
complex[1][0][0] = complex;
calculateSum(complex) // === 4
```
~~~

```html
<script src="https://cdnjs.cloudflare.com/ajax/libs/jsPlumb/2.11.2/js/jspl
<link id="box-and-pointer-style" rel="stylesheet" href="https://cdn.jsdeli
<script src="https://cdn.jsdelivr.net/gh/TheUnlocked/box-and-pointer@maste

<script>

function visualize(n) {
    let ct = 0;
    const refs = new Map();

```

```js
function calculateSum(ll) {
    return 0;
}

visualize([1, [2, [3, null]]]);

const complex = [1, [[-1, null], [3, null]]];
complex[1][0][0] = complex;
visualize(complex);

```

```ts
type LL = [LL, LL] | number | null;
declare function calculateSum(ll: LL): number;

check(calculateSum([1, null]) === 1, SHOW_TEST
check(calculateSum([1, 1]) === 2, SHOW_TEST);
check(calculateSum([1, [2, [3, null]]]) === 6,
check(calculateSum([null, 1]) === 1, SHOW_TEST

const complex = [1, [[-1, null], [3, null]]] a
complex[1][0][0] = complex;
check(
    calculateSum(complex) === 4,
    'Some linked lists structures may have cyc
);

```

Your tests compiled successfully!

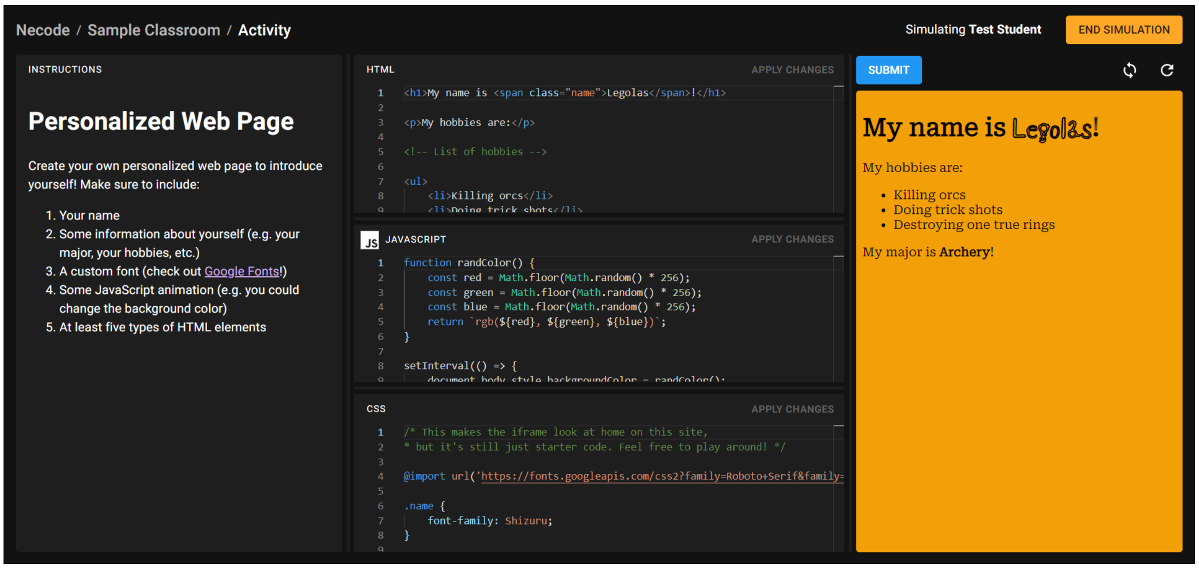
Figure 2.6: *A sample configuration page for an activity*

an activity. However, from the developer perspective, “activity” more often refers to the *type* of activity from which instances are configured. To be entirely unambiguous, one should use the terms “activity instance” and “activity type” to refer to the two concepts, though we will usually not do so in the rest of this thesis when we believe context is a sufficient disambiguator.

2.3.3.1 HtmlTestActivity

Prior to this thesis there were already a few built-in activity types, though most of them (excluding the text widget and the canvas ring in Section 2.3.4) were derived from a single “base” activity type: the HtmlTestActivity. The same core code is used for the activity seen and configured in Figures 2.4 and 2.6, as well as for some others like a p5.js activity and even an activity where students can write GLSL shaders.

The HtmlTestActivity code is capable of, as its name might suggest, rendering HTML and running tests. Depending on how it is configured, students may write their own HTML like in Figure 2.7, or, as was the case in Figure 2.4, the instructor may hide the HTML from them and just have students write in a selected language. The instructor may also write tests in a special TypeScript DSL during configuration which are automatically run when a student tries to submit their code to the instructor. The submission system is built into Necode, the testing DSL is just part of the activity.



Personalized Web Page

Create your own personalized web page to introduce yourself! Make sure to include:

1. Your name
2. Some information about yourself (e.g. your major, your hobbies, etc.)
3. A custom font (check out Google Fonts!)
4. Some JavaScript animation (e.g. you could change the background color)
5. At least five types of HTML elements

```html
<h1>My name is <span class="name">Legolas</span>!</h1>

<p>My hobbies are:</p>

<!-- List of hobbies -->

<ul>
    <li>Killing orcs</li>
    <li>Doing trick shots</li>
```

```javascript
function randColor() {
    const red = Math.floor(Math.random() * 256);
    const green = Math.floor(Math.random() * 256);
    const blue = Math.floor(Math.random() * 256);
    return `rgb(${red}, ${green}, ${blue})`;
}

setInterval(() => {
    document.body.style.backgroundColor = randColor();
```

```css
/* This makes the iframe look at home on this site,
* but it's still just starter code. Feel free to play around! */

@import url('https://fonts.googleapis.com/css2?family=Roboto+Serif&family=

.name {
    font-family: Shizuru;
}
```

My name is Legolas!

My hobbies are:

- Killing orcs
- Doing trick shots
- Destroying one true rings

My major is **Archery**!

Figure 2.7: *An activity in which a student needs to write their own HTML*

2.3.4 Real-Time Communication

Necode activities can use peer-to-peer real-time communication between students, where any kind of data can be sent from plain text to video. In Figure 2.8,[3] the “canvas ring” activity is using Necode’s RTC to send the visual output of one user’s code to the next, who draws on top of that incoming image and sends it to the next user in a loop.

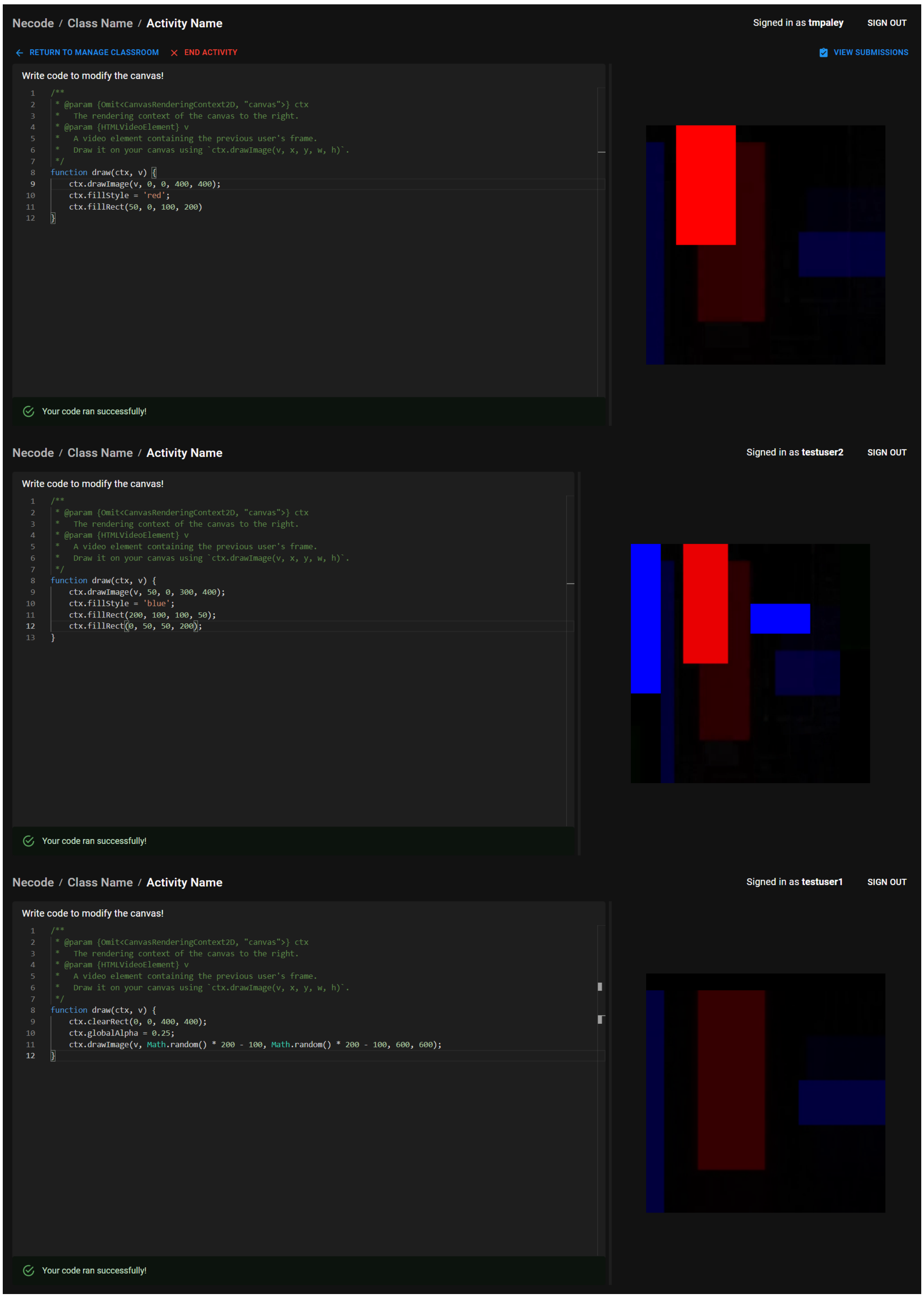
In order to create the necessary peer-to-peer connections, Necode has programmatic “policies” which are algorithms that tell clients to create and destroy peer-to-peer connections (i.e. construct the P2P network topology) in response to events like a user joining or leaving an activity. The aptly-named “ring” policy that powers the canvas ring instructs clients to create peer-to-peer connections such that users are sending data to each other in a directional ring network. These policies are defined independently of activities, so activity developers can change which policy they want an activity to use simply by putting a different policy name.

2.3.5 Languages

Necode has the potential to support many languages as long as they can be implemented (or at least somehow invoked from) JavaScript, similar to the Racket implementation requirement of DrRacket (Section 2.2.2). Again drawing parallels to DrRacket, these languages are somewhat independent of activities, with activities and languages being automatically linked through shared “features” rather than DrRacket’s capabilities (see Section 3.2 for more information on features in Necode).

However, unlike DrRacket, because Necode allows different activities to have completely different user interfaces, the entire interface can be tailored for a particular language if desired rather than simply having extra UI elements appear or disappear depending on what the language can support. At the same time, more generalized activities can still support many different languages, provided the language supports all the features that the activity needs.

[3]This particular screenshot was captured before this thesis, so the user interface is slightly different.


Necode / Class Name / Activity Name
Signed in as tmpaley
SIGN OUT
RETURN TO MANAGE CLASSROOM
END ACTIVITY
VIEW SUBMISSIONS

Write code to modify the canvas!

```
/**
 * @param {Omit<CanvasRenderingContext2D, "canvas">} ctx
 *   The rendering context of the canvas to the right.
 * @param {HTMLVideoElement} v
 *   A video element containing the previous user's frame.
 *   Draw it on your canvas using `ctx.drawImage(v, x, y, w, h)`.
 */
function draw(ctx, v) {
    ctx.drawImage(v, 0, 0, 400, 400);
    ctx.fillStyle = 'red';
    ctx.fillRect(50, 0, 100, 200)
}
```

Your code ran successfully!

Necode / Class Name / Activity Name
Signed in as testuser2
SIGN OUT

Write code to modify the canvas!

```
/**
 * @param {Omit<CanvasRenderingContext2D, "canvas">} ctx
 *   The rendering context of the canvas to the right.
 * @param {HTMLVideoElement} v
 *   A video element containing the previous user's frame.
 *   Draw it on your canvas using `ctx.drawImage(v, x, y, w, h)`.
 */
function draw(ctx, v) {
    ctx.drawImage(v, 50, 0, 300, 400);
    ctx.fillStyle = 'blue';
    ctx.fillRect(200, 100, 100, 50);
    ctx.fillRect(0, 50, 50, 200);
}
```

Your code ran successfully!

Write code to modify the canvas!

```
/**
 * @param {Omit<CanvasRenderingContext2D, "canvas">} ctx
 *   The rendering context of the canvas to the right.
 * @param {HTMLVideoElement} v
 *   A video element containing the previous user's frame.
 *   Draw it on your canvas using `ctx.drawImage(v, x, y, w, h)`.
 */
function draw(ctx, v) {
    ctx.clearRect(0, 0, 400, 400);
    ctx.globalAlpha = 0.25;
    ctx.drawImage(v, Math.random() * 200 - 100, Math.random() * 200 - 100, 600, 600);
}
```

Your code ran successfully!

Figure 2.8: *Three users interacting in a canvas ring activity*

Chapter 3

Methodology

In this chapter, we will describe in detail all of the APIs and abstractions we have developed for extending Necode, as well as the rationale behind our design decisions. To do this, we have divided this chapter into four sections. The first three will discuss our APIs and abstractions for each type of module (activities, languages, and policies), and the last will describe how they all come together in the form of a plugin.

3.1 Activities

Activities make up the heart of Necode, and because they are the part students and instructors most directly interact with, providing effective tools and abstractions to third-party developers is essential to our goal of building extensibility. In this section, we will describe the various APIs we have created for third-party developers to create their own activities for Necode, and the rationale for the design decisions we made in producing those APIs. We will begin with the discussion of general activity creation and configuration, then move onto real-time communication features, and finally talk about the tools we provide for building activity user interfaces.

3.1.1 Creating Activities

In order to create an activity, third-parties must first define an `ActivityDescription` (Figure 3.1) for their activity. This contains various fields such as the activity ID and display name, a React component containing the activity itself, and information about what language features the activity requires and which RTC policies it wants to use. This is then registered in the `registerActivities` plugin hook (see Section 3.4.1) by calling `ActivityManager.registerActivity`.

```
export interface ActivityDescription<
    ConfigData = any,
    Features extends readonly Feature[] = any[]
> {
    id: string;
    displayName: string;

    defaultConfig: ConfigData;
    requiredFeatures: Features;
    configurePolicies?: (config: ConfigData) => readonly PolicyConfiguration[];

    activityPage: Importable<
        ComponentType<ActivityPageProps<Features, ConfigData>>
    >;
    configWidget?: Importable<
        ComponentType<ActivityConfigWidgetProps<ConfigData>>
    >;
    configPage?: Importable<
        ComponentType<ActivityConfigPageProps<Features, ConfigData>>
    >;
}
```

Figure 3.1: *The `ActivityDescription` interface*

Because activity components can be quite large, especially when external packages are used, Necode loads the components lazily on demand using the `Importable<T>` type (which is just an asynchronous thunk of `T`). Combining this with dynamic `import()` allows for easy code splitting and avoids needing to download the code for every activity in every plugin (or even every activity in a single plugin) when an instructor or student is only using a single activity. An example of this can be seen in Figure 3.2, where `'./CanvasActivity.ts'` exports a React component called `CanvasActivity`.

3.1.2 Configuration

Most activities are not ones where instructors create an instance and have everything set up right out of the box. Configuring properties like instructions, sample code, and so on will be necessary to tailor a generalized activity type for a particular lesson or concept the instructor is trying to teach.

Configuration can be represented as any JSON-serializable object, with default values provided in the `ActivityDescription.defaultConfig` field. This value will then be provided as the `activityConfig` prop in the `ActivityDescription.activityPage` React component so that an activity developer can change any part of their activity based on this user configuration.

```
const canvasActivityDescription = activityDescription({
    id: 'core/canvas-ring',
    displayName: 'Canvas Ring',
    requiredFeatures: [
        'entryPoint/any'
    ],
    configurePolicies: () => [{ name: 'ring' }],
    activityPage: async () => (await import('./CanvasActivity')).CanvasActivity,
    configWidget: async () => (await import('./Widget')).CanvasWidget,
    defaultConfig: { canvasWidth: 400, canvasHeight: 400 } as Configuration,
});
```

Figure 3.2: *The `ActivityDescription` for the "canvas ring" activity*

Changing configuration requires additional UI elements however. Sometimes configuration will be changed in a widget in the lesson management page (Figure 3.3) and sometimes, as discussed in Section 2.3.2, configuration will be changed in a full-screen configuration page. These are declared by the activity developer in the optional `ActivityDescription.configWidget` and `ActivityDescription.configPage` fields respectively. These fields should provide importable components which are "controlled" by their `activityConfig` and `onActivityConfigChange` props. A simplified version of the interfaces defining these sets of properties is shown in Figure 3.4.

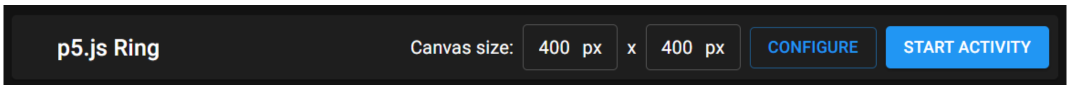


Figure 3.3: *Screenshot of a custom widget*

3.1.3 Abstracting Real-Time Communication

One of Necode's key features is peer-to-peer real-time communication via WebRTC, but facilitating that communication was difficult prior to this thesis. Establishing and maintaining WebRTC links requires care, and anything beyond toy examples tends to involve a significant amount of engineering to make sure that the right data is being sent and received at the right time [20].

Using existing libraries to simplify the WebRTC interface helps (and we use one called `simple-peer`[1]), but there are limits to how far they can take us. Because we are integrating this new API into an existing system and integrating it with other abstractions (namely policies, see Section 3.3), we require considerable control over how peers are connected to each other. At the same time, we still need to ensure that the API third-party developers will use is ergonomic

[1]https://github.com/feross/simple-peer

```
export interface ActivityConfigPageProps<Features, ConfigData> {
    id: string;
    classroomId: string;

    language: LanguageDescription;
    features: FeatureObject<Features>;

    activityConfig: ConfigData;
    onActivityConfigChange: (newConfig: ConfigData) => void;
}

export interface ActivityConfigWidgetProps<ConfigData> {
    id: string;
    classroomId: string;

    activityType: ActivityDescription;
    dragHandle: RefConnector;
    goToConfigPage: (() => void) | undefined;
    startActivity: () => void;

    activityConfig: ConfigData;
    onActivityConfigChange: (newConfig: ConfigData) => void;

    displayName: string;
    onDisplayNameChange: (newDisplayName: string) => void;
}
```

Figure 3.4: *Simplified interfaces for configuration page and widget component props*

for React and does not expose unnecessary details. Some existing libraries provide one of those requirements, none provide both.

To provide for our use case, we built a new abstraction over WebRTC in which developers declare communication networks which consist of many channels that are created in a declarative ad-hoc manner using React hooks.

3.1.3.1 Media Channels

One of the big advantages of using peer-to-peer communication like WebRTC over proxying through a server is that audio and video data (media streams) can be transmitted more efficiently and with less server-side strain. Necode previously used video transmission over WebRTC in the “Canvas Ring” activity (Section 2.3.4), and our experience from implementing that guided the design of Necode's new API.

In our design, media streams are sent between peers using a hook superficially similar to React's built-in `useState`, but in which setting the state sends the media stream to linked peers, and the current value is an array of all incoming media streams being received from linked peers. Note that peer linking is directional, so depending on how links have been established per the current RTC policy (Section 3.3), a hypothetical User A could send a media stream to User B but receive media streams from User C and User D.

This hook is `useMediaChannel`, which takes in a network ID (Section 3.1.3.4) and a developer-specified channel name. This channel name is arbitrary and is simply used to ensure that clients can coordinate which data should be associated with which hooks. A toy example of `useMediaChannel` in action can be seen in Figure 3.5.

```
const [inboundMediaStreams, setOutboundMediaStream] =
    useMediaChannel(NetworkId.NET_0, 'canvas');

function canvasLoadHandler(canvas: HTMLCanvasElement | null) {
    if (canvas) {
        setOutboundMediaStream(canvas.captureStream(10));
    }
}

return <>
    <canvas ref={canvasLoadHandler} />
    <Stack>
        {inboundMediaStreams.map(s =>
            <Video key={s.id} srcObject={s} />
        )}
    </Stack>
</>;
```

Figure 3.5: *An example of how `useMediaChannel` might be used*

Because `useMediaChannel` only provides the media streams that peers send, it offers no mechanism for determining which peer provided which media stream. However, by using data channels (described in the next section), peers can send additional data about themselves along with the ID of the media stream they sent so that recipients can cross-reference data from the media and data channels.

3.1.3.2 Data Channels

Data channels, which allow peers to send strings or byte sequences to each other, behave more imperatively than media channels. There are two hooks that can be used to create a data channel, `useDataChannel` and `useStringDataChannel`, and rather than returning an array tuple with incoming data and a function to send outgoing data, they return a single "emit" function used to send `Uint8Arrays` or strings respectively to linked peers. To get data in, they take a third argument (after the network ID and channel name) which is a callback for when data is received, as demonstrated in Figure 3.6. For more controlled communication, the emit function has an optional argument which can send messages only to specific peers rather than to all linked peers.

```
const [messages, setMessages] = useState<{
    content: string,
    user: string,
}[]>([]);

const sendMessage = useStringDataChannel(
    NetworkId.NET_0,
    "chat",
    (content, from) => {
        setMessages(m => [...m, { content, user: from.displayName }]);
    }
);

return <>
    {messages.map((msg, i) =>
        <div key={i}>{msg.user}: {msg.content}</div>
    )}
    <input onKeyUp={e => {
        if (e.key === 'Enter') {
            sendMessage(e.currentTarget.value);
            e.currentTarget.value = "";
        }
    }} />
</>;
```

Figure 3.6: *An example of how `useStringDataChannel` might be used*

In certain cases, simply sending and receiving data is insufficient, and an activity will actually want to know when a peer-to-peer connection is established or broken. For this, there is a more complex `use(String)DataChannelLifecycle` hook whose callback will additionally be be fired when a peer connects/disconnects, and will include a parameter indicating what kind of event (connect/disconnect/message) occurred. In Figure 3.6, this could be used to add a system message when peer connects or disconnects.

Note that naive attempts to implement this data channel abstraction may fail due to a Firefox bug which makes it impossible to reliably create multiple data channels on a single WebRTC peer connection [21]. To allow developers to use multiple data channels in their activities, Necode needs to perform its own data channel multiplexing keyed by a hash of the channel name.

3.1.3.3 Shared Documents

While media channels and data channels are sufficient primitives to communicate any data that is desired, they are still sometimes too low-level to be convenient in practice. In particular, creating shared documents would be challenging even if possible for third parties to implement on their own just using `useDataChannel`. While we can't provide specialized APIs for every RTC feature third-party developers might want, shared documents are particularly relevant because of the ubiquity of shared code editors in educational programming environments.

Fortunately for us, we do not need to develop our own concurrent editing implementation. Yjs is an open-source JavaScript library which provides shared documents using Conflict-free Replicated Data Types (or CRDT) [16], and which we can fairly easily integrate with `useDataChannel` before exposing to developers in a `useY` hook. Because of how Yjs propagates updates, users do not need to be directly linked to each other to share an editor as long as they are connected transitively, meaning shared documents will work even with policies which create a minimal number of links between users.

A tricky aspect of using Yjs is that document state changes occur through internal mutation which lives outside of the reactive world. To mitigate this, `useY` does not return a Yjs document directly but instead returns a reactive handle around the document. This does not matter quite as much for `useY` since the lifetime of a Yjs document is the same as the lifetime of the handle, but the `useYText` hook, which consumes a document handle and returns a handle around `Y.Text`, will cause reactive updates whenever the text content changes. The original Yjs objects are still accessible when necessary, but this wrapping should help avoid `useY` being a reactivity footgun.

Necode also provides a `useYInit` hook for synchronizing document state initialization (e.g. to provide starter code), a `useYAwareness` hook to synchronize "awareness" information (see Section 3.1.5.3), and a few other utilities to transform the Yjs document.

3.1.3.4 Networks

The RTC hooks do not specify which peers to connect to and instead implicitly obey the connections that the current policy dictates. However, in some cases it can be necessary to have multiple concurrent policies with different channels on each policy. For example, a developer might want to make an activity in which students are broken up into breakout rooms but can also ask questions in a class-wide chat room. Without multiple policies, such a thing would be effectively impossible.

The `configurePolicies` field of an activity's `ActivityDescription` allows developers to specify an array of policies to use in their activity, and that array can include multiple policies. The indices of policies in that array map directly to the values in the `NetworkId` enum when used in RTC hooks (i.e. element 0 maps to `NetworkId.NET_0`, element 1 maps to `NetworkId.NET_1`, etc.).

Note that unlike channels which can be declared in an ad-hoc manner with hooks, networks cannot be. Channels can be handled purely on a peer-to-peer level, but networks require the signaling server to be "in on it" so that it can link users according to the specified policy. While it is theoretically possible to detect all networks declared using hooks directly within an activity component, it would not always be possible to detect networks declared using hooks in children. Without being able to statically analyze the necessary networks, networks cannot necessarily be formed since only the instructor is privileged to instruct Necode to create a network, and the hooks that are invoked may be different between users.[2]

3.1.4 Persistent Student Submissions

For many purposes, transient data stored only on users' local sessions is perfectly acceptable. For example, while it may be inconvenient, there is relatively little danger in a small program written for an in-class activity being lost if a user closes or refreshes the page. However, if a single user refreshing caused loss of data for everyone, that would be more problematic. This situation would occur if an activity had students submit data, such as code, to the instructor via `useDataChannel`, and then the instructor refreshed their page.

Prior to this thesis, Necode had a persistent submission system in which data could be saved as a submission using a callback prop, and then loaded into the activity with another activity prop which had to be reset with yet another callback prop after the submission was done being loaded. This system was highly fragile and challenging to develop with.

To align with the RTC API, submissions now use a hook, `useSubmissions`, which accepts a callback for when a submission is loaded and returns a function to submit an object. This is very similar to the interface for `useDataChannel`, only submitted content is saved in a database and loaded on demand from the instructor. A positive side-effect of this shift is that Necode can now detect whether or not an activity uses submissions by whether or not the `useSubmissions` hook is called. Previously, even if an activity did not make use of submissions, the UI element to view student submissions would still appear when instructors loaded that activity. By being able to detect whether an activity will make use of submissions, Necode can dynamically alter the activity UI accordingly. We believe that this model of using hook presence to indicate feature usage could be beneficial to designing new features for Necode, and designing extensibility for other React applications.

[2]Because channels are given a name, it is actually okay if one peer thinks a channel exists while the other peer does not. The peer that is not aware of the channel will just discard anything it receives from unknown channels as invalid while attempting to route the communication in the multiplexer.

3.1.4.1 Submission Versioning

If an activity changes over time, the data which is stored in submissions may change as well.[3] This is problematic since submissions to old activities could break, causing errors when an instructor attempts to load one. While it is unlikely that an instructor would go back to a previously-run activity after an update, the design should still account for it.

To mitigate this issue, we introduced submission versioning, in which each submission is tagged with an integer `version` field indicating the version of the submission schema being used. This is readable when a submission is loaded and is well-typed using TypeScript's discriminated union type narrowing.

3.1.5 Abstracting the User Interface

It can be difficult and time-consuming for third-party developers to create the user interface for a new activity completely from scratch, so we expose some libraries to assist in building UI. First, plugins are designed such that developers can use the `@mui/material` UI library which Necode itself uses for all of its user interface, and which provides significant scaffolding on top of vanilla HTML and JavaScript.

But not all of Necode's UI comes straight from MUI; there are also a number of custom-built UI components and visual elements which third parties may want to use in their own activities. For this, we created the `@necode-org/activity-dev` package which contains certain useful components to make third-party activities look and feel like first-party activities.

3.1.5.1 Panes

Most of the components we provide are simple visual elements or controlled inputs, and are not worth calling out by name here. However, our `Panes` components are more interesting.

Very often, programming environments will have have two or more side-by-side resizable "panes" showing the user various kinds of content such as their code, their output, perhaps a file directory structure—the details vary between applications. However, this feature is common between nearly every programming environment, whether its purpose is for industry or for education. To make this claim and justify the value of designing a new component for this purpose, in Figure 3.7 we examined the top 20 offline and top 10 online IDEs[4] to see whether they included "pane" UI elements.

Due to how prevalent panes are in editors, designing an API for developers to be able to use panes with minimal effort is important. There exist libraries to do this in React already,

[3]This is also true for activity configuration, but applying the same technique to activity configuration that we apply to submissions would be somewhat tricky. We instead expect developers to handle invalid configurations on their own and assume defaults where a field is missing. Because configuration changes are often adding new fields while submission format changes are more likely to change the values of fields, we expect breaking schema changes to be a larger issue with submissions.

[4]Based on the March 2023 standings from the Google Trends-based PYPL IDE/ODE indices [22, 23].

Offline IDE	Panes?	Online IDE	Panes?
Visual Studio	✓	JSFiddle	✓
Visual Studio Code	✓	Koding	✓
Eclipse	✓	Codio	✓
pyCharm	✓	PythonAnywhere	✗
Android Studio	✓	DartPad	✓
IntelliJ	✓	Ideone	✗
NetBeans	✓	Repl.it	✓
RStudio	✓	Cloud9	✓
Sublime Text	✓	Goorm	✓
Atom	✓	Codeanywhere	✓
Xcode	✓		
Code::Blocks	✓		
Vim	✓*		
PhpStorm	✓		
Xamarin	✓		
Komodo	✓		
Qt Creator	✓		
geany	✓		
Emacs	✓		
JDeveloper	✓		

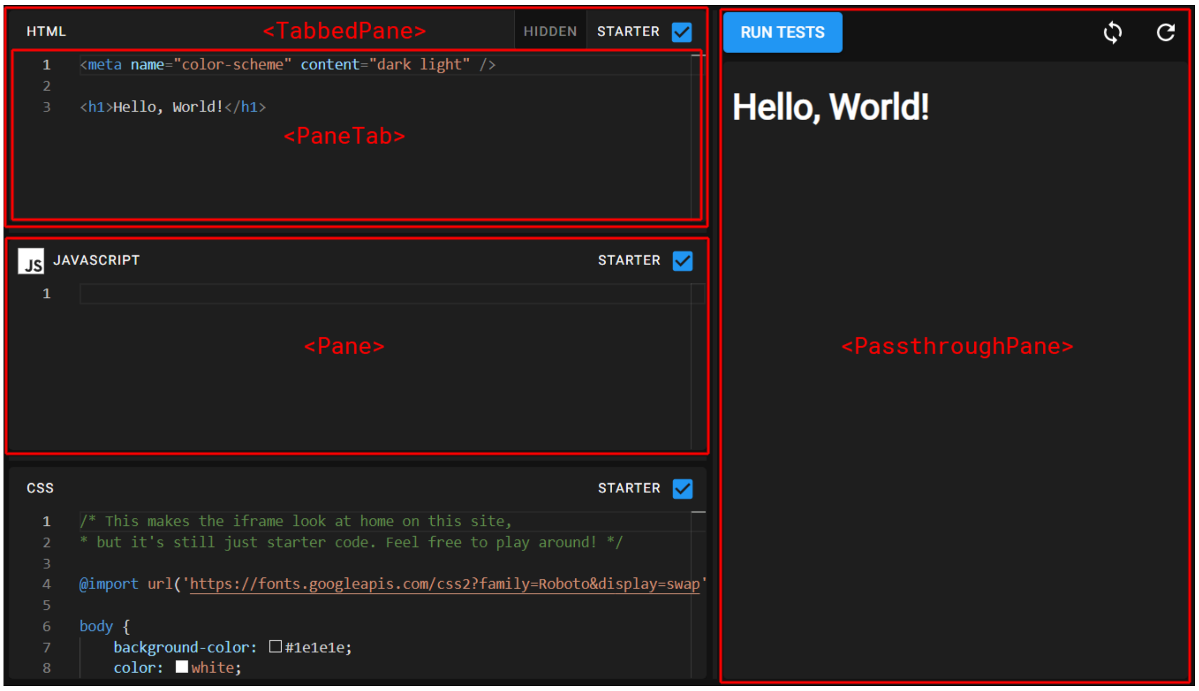
* Supports "splits" which can be utilized by plugins

Figure 3.7: *A comparison of various programming environments by whether or not they use UI panes*

such as `react-split-pane` and `react-reflex`, but they are not well-suited for dynamic and responsive layouts as are often seen in IDEs due to the difficulty of programatically constructing a nested structure they require for both vertical and horizontal resizable panes. By designing our own components, we can both standardize the visual style of panes in Necode and simplify the element structure for easier development.

To do this, we separate pane arrangement from the pane elements. Within a `<Panes>` element, every pane is placed in a flat structure, in order for where the pane should appear from top-to-bottom, then left-to-right. There are three types of pane that a developer can use depending on their use case (Figure 3.8):

- `Pane`– This pane displays its content inside a card with a header based on various component props.
- `TabbedPane`– This is like a regular pane, except its children consist of `<PaneTab>` elements which can be switched between using a "tab" interface at the top of the pane.
- `PassthroughPane`– This pane simply displays its content without any additional styling.



```html
<meta name="color-scheme" content="dark light" />

<h1>Hello, World!</h1>
```

```css
/* This makes the iframe look at home on this site,
 * but it's still just starter code. Feel free to play around! */

@import url('https://fonts.googleapis.com/css2?family=Roboto&display=swap'

body {
    background-color: #1e1e1e;
    color: white;
```

Hello, World!

Figure 3.8: *A showcase of the various types of pane*

Then to indicate how these panes are arranged, the `layouts` property is set on the `<Panes>` element providing (for multiple screen sizes if desired), an array representing how many panes should be stacked in each column. For example, the layout in the in Figure 3.8 would be `[3, 1]`.[5] If a pane (or `PaneTab`) should be hidden for any reason, it can be marked with the `hidden` property to prevent it from displaying while still considering it for the layout. When every pane in a column is hidden, the entire column is removed.

3.1.5.2 Widgets

Widgets, as mentioned previously, provide a way for instructors to configure activities right from the lesson management page. While an activity's widget can be completely custom-made, most activities will not require that much customization, and requiring widgets to be built from scratch would be burdensome to most activity developers. To simplify the process of creating widgets, `@necode-org/activity-dev` contains two relevant exports:

- `DefaultActivityWidget`– The default widget used when no widget component is provided in an `ActivityDescription`, though it accepts a `children` prop for additional inputs.
- `ActivityWidgetBase`– A stripped-down widget which contains no inputs or buttons right of the activity title. A custom user interface can be provided in the `children` prop.

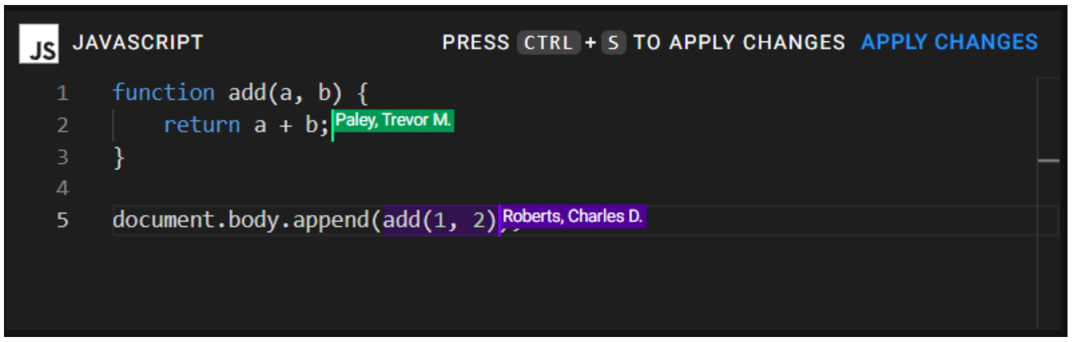
[5] A slightly more complicated object structure can be used to weight how large each pane should be by default.

These enable third-party developers to easily put light configuration into their widgets without needing to re-implement any UI to maintain consistency with the rest of Necode.

3.1.5.3 Code Editors

For code editing, Necode uses Microsoft's Monaco editor,[6] the same editor which powers Visual Studio Code [24]. This works extremely well, but when combined with real-time communication for shared editors, some issues start to arise. While the `y-monaco` package exists to sync the editor with a Yjs document (created by `useY`, see Section 3.1.3.3), there are several nuances to that regarding bundling, line endings, and the undo stack. Because these issues are non-obvious and finicky to solve, we provide our own `Editor` component to wrap Monaco with shared document support.

In addition to simply making shared documents work properly with Monaco, we also support "awareness" features (Figure 3.9) if the developer provides an awareness object from the `useYAwareness` hook. While a similar effect is used in Visual Studio Code's first-party Live Share extension [25], that functionality is not included in Monaco and so we had to build our own awareness UI which is also included in our `Editor` component.



```
function add(a, b) {
    return a + b;
}

document.body.append(add(1, 2));
```

Figure 3.9: *A screenshot of two users collaborating with Necode's `Editor` component*

3.2 Languages and Features

Multi-language support is a major goal of Necode, but the wide variety of activities that Necode *could* support makes that harder to achieve than simply having a single endpoint for activity developers that runs an interactive shell.[7] For example, one activity might want to load a program directly in the global context, letting it define variables and functions right on the JavaScript `window` or `globalThis` object, whereas another might want to access a single entry point which is isolated from and has no impact on the global environment.

[6]Provided by the `@monaco-editor/react` package.

[7]Remember that Necode runs code client-side as opposed to the more abstracted server-side execution that other multi-language environments prefer.

3.2.1 Old Feature Design

To support multiple different modes of execution, Necode used a concept called "features," which allowed languages to declare what modes of operation they were capable of, and allowed activities to declare what modes of operation were required. When Necode found a match, it would make the matching language available for the matching activity.

To give an idea of how this would look in practice, we can look at the old language description (an object describing various properties of the language, including its features) for JavaScript in Figure 3.10. Each feature used in the language would be defined with a field for its name, plus some type annotations to describe how the feature would be used. For example, Figure 3.11 shows the implementation of the `supports:entryPoint` feature, which indicates that the language can support requesting a single entry point as opposed to just running the code in the global context (which is the `supports:global` feature).

```
export const javascriptDescription = languageDescription({
    name: 'javascript',
    monacoName: 'javascript',
    displayName: 'JavaScript',
    icon: JavascriptIcon,
    features: [
        supportsEntryPoint,
        supportsGlobal,
        supportsIsolated,
        supportsBabelPlugins
    ] as const
});
```

Figure 3.10: *The JavaScript language description from before this work*

```
const supportsEntryPoint: FeatureDescription<{
    entryPoint: string;
}> = {
    name: 'supports:entryPoint'
};
```

Figure 3.11: *The `supports:entryPoint` feature description from before this work*

Then in the language implementation, which was separate from the language definition,[8] there would be a single function to generate JavaScript source that could be `eval()`'ed or placed in a `<script>` tag by activities as necessary. This function would be provided an object representing some subset of the fields defined in the required feature description types, as seen in the JavaScript language implementation in Figure 3.12.

[8]This separation was a surprisingly good idea which went away for quite a long time while designing the language abstractions for this thesis, only to return closer to the end when the new language/feature design materialized.

```
export class Javascript implements RunnableLanguage<...> {
    toRunnerCode(code: string, options: FeatureOptionsOf<...>) {
        try {
            const result = transformSync(code, {
                plugins: [
                    ...options.babelPlugins ?? [],
                    transformPreventInfiniteLoops[9]
                ]
            });
            if (options.global) {
                return result!.code!;
            }
            if (typeof options.entryPoint === 'string') {
                return ...;
            }
            throw new Error(
                'Javascript code must be generated in' +
                'either global or entryPoint mode'
            );
        }
        catch (e: any) {
            // The code will throw some sort of syntax error anyways,
            // so we'll let it throw the browser version.
            return code;
        }
    }
}
```

Figure 3.12: *A simplified version of the implementation of the JavaScript language from before this work*

While that sounds nice in concept, a few seconds with it in practice shows some clear flaws. Firstly, in order for activities to invoke the language implementation with any subset of supported features, the language must support being invoked with *every* subset of features, meaning the complexity of the language implementation could as much as double for every new feature added. Of course, some combinations of features either do not make any sense (e.g. generating code which both provides an isolated entry point and runs everything in the global context) or simply cannot be implemented together given current technology, resulting in a potentially undocumented and certainly unenforced set of feature combinations which will fail at runtime.

Beyond these issues however, there are also some more fundamental design limitations. For example, some languages may not be able to easily have their entire runtime compressed down into a single string. Other languages may not be able to communicate synchronously, and would

[9]This is explained in Section 3.2.2.2 of the original MQP report [9].

need a way of specifying that activities will need to support asynchronous communication. Perhaps all existing features could be reinterpreted as being the asynchronous version of themselves with a new `supports:sync` feature for languages like JavaScript which do support synchronous execution, but what if a language only supports synchronous execution for certain features (e.g. it supports a synchronous entry point but not synchronously loading into the global context)? And what if a language cannot directly interface with JavaScript objects and can instead only talk through strings?

There are an extraordinarily large number of these "what-if"s, and accounting for all of them ahead of time is impossible. For third-party languages that cannot support all of the built-in scenarios, their only option is to create new bespoke features that work for their use case. The issue with that? Existing languages won't support them.

3.2.2 Trait-Based Features

These issues may be new in the world of educational programming environments, but they are not new in other types of software. In fact, programming languages themselves have thoroughly explored this territory before, and have developed solutions which we can (and did) use to improve features in Necode.

Using the presence of fields in an object argument to switch between different implementations in the `toRunnerCode` method is really just a clunky form of message passing, which could be replaced with the interface abstraction. Rather than each feature declaring what fields of the object it provides, features would declare what methods the language implementation must support, replacing the single `toRunnerCode` method with a whole collection of feature methods. Note that the intersection between features is lost in this design—having `supports:isolated` for iframe/worker support will no longer result in any change to the methods provided by `supports:global`—but that's okay. After all, supporting `isolated` and `global` together should not require that the language supports `isolated` and `entryPoint` together. Languages *should* opt in to each specific combination that they can perform.

But converting features to interfaces, while a big step forward, does not solve all of our issues. Interfaces as seen in languages like Java and C# do not allow existing classes (languages implementations in our case) to implement new interfaces (features).[10] For that, we need to separate features from the language definition entirely and change our abstraction from interfaces to the traits and typeclasses seen in languages like Rust and Haskell.

3.2.2.1 Language Features

Trait-based features (which will just be called "features" from now on) are declared statically without any need for runtime information via TypeScript's module augmentation and declaration

[10] At least for now. For C#, see https://github.com/dotnet/csharplang/discussions/5496.

merging, as seen in Figure 3.13.[11] This declaration is used to ensure the well-typed-ness of other code involving features, though as mentioned it has no runtime effect. Language descriptions too have no runtime knowledge of a language's supported features when they are registered in a plugin's `registerLanguages` hook (Figure 3.14, see Section 3.4.1 about hooks). All feature information is instead introduced in the `registerFeatures` plugin hook, where the `FeatureManager.implementFeature` method can be used to implement a feature for a particular language (Figure 3.15). This is analogous to Rust's `impl ... for ...` or Haskell's `instance ... where ...` syntax forms.

```
declare module '@necode-org/plugin-dev' {
    interface FeatureMap {
        'evaluate/any': {
            evaluate<T>(code: string): Promise<T>;
        };
    }
}
```

Figure 3.13: *An example feature declaration in Necode*

```
manager.registerLanguage({
    name: 'javascript',
    monacoName: 'javascript',
    displayName: 'JavaScript',
    icon: JavascriptIcon,
});
```

Figure 3.14: *The JavaScript language description being registered in the Core plugin.*

```
manager.implementFeature('javascript', 'evaluate/any', [], async () => ({
    async evaluate(code) {
        return eval(
            '"use strict";(' +
            (await transformAsync(code, {
                plugins: [babelPluginTransformPreventInfiniteLoops]
            }) ?? code) +
            ')'
        );
    },
}));
```

Figure 3.15: *An implementation of the `evaluate/any` feature for JavaScript*

For convenience, and to help make code splitting easier, multiple features can be implemented for a language at once with `FeatureManager.implementFeatures` (note the "s") where the

[11]The `evaluate/any` feature shown in the figure is included in Necode out of the box so it does not actually use module augmentation in practice, though it would be functionally equivalent if it did.

implementation callback returns an object whose keys are the feature names and whose values are the respective implementations. The actual `FeatureManager.implementFeatures` call for JavaScript is shown in Figure 3.16, where the dynamic import allows Babel to be loaded only when the JavaScript language is used in an activity. Importantly, this still does not preclude other plugins from implementing individual features, or even whole sets of features, for JavaScript later.

```
manager.implementFeatures(
    'javascript',
    [
        'entryPoint/any/sync',
        'evaluate/any/sync',
        'evaluate/any',
        'iframe/static',
        'worker/static',
        'repl/instanced/fullSync',
        'js/babel',
    ],
    [],
    async () => (await import('./languages/javascript')).default
);
```

Figure 3.16: *The feature implementation code for JavaScript*

3.2.2.2 Using Features

When an activity component loads, its props will populate with a `features` field, containing the "feature object" representing all of the feature implementations that activity requested. The structure of the feature object is defined by the feature path. The slashes in a path become (well-typed) dereferences of the feature object. For example, following the feature declaration in Figure 3.13, `featureObj.evaluate.any` would map to the object containing the `evaluate` method. This "namespacing" helps avoid name collisions between similar features while ensuring that the abstraction is TypeScript-friendly.

3.2.2.3 Free Features

The keen-eyed readers may have noticed earlier that in Figure 3.16, JavaScript implements `evaluate/any` and `evaluate/any/sync`, but with entry points only implements `entryPoint/any/sync`. This is because for entry points, "sync-ness" refers to whether the entry point itself can be executed synchronously rather than whether the code to generate it can be.[12] If the evaluation is asynchronous anyways, it can use Babel's `transformAsync` API

[12]If asynchronous entry point creation is required and the entry point itself is asynchronous, the language can just return an asynchronous entry point which first waits for the the true entry point to be created, then

instead of the synchronous version, but that does not apply if the entry point function needs to be obtained synchronously either way.[13]

However, if there is an implementation of `entryPoint/any/sync` for JavaScript, it would be silly for an activity wanting `entryPoint/any` to not be able to use JavaScript due to lack of feature support. And needing to duplicate the implementation with only minor alterations to make the generated entry point asynchronous would be cumbersome and error-prone. Fortunately, Necode allows implementing features in terms of other features. In Figure 3.17, we can see that the third argument to `FeatureManager.implementFeature` is a list of features whose implementations are given to the implementation of the specified feature as a feature object, as described in 3.2.2.2.[14]

```
manager.implementFeature(
    'javascript',
    'entryPoint/any',
    ['entryPoint/any/sync'],
    async obj => ({
        entryPoint: async code =>
            obj.entryPoint.any.sync.entryPoint(code),
    })
);
```

Figure 3.17: *A dependent feature implementation for JavaScript*

However, this new feature implementation has nothing to do with JavaScript. It should work for any language which has an implementation for `entryPoint/any/sync` but does not have an implementation for `entryPoint/any`, but it is being artificially limited to JavaScript by specifying it as the implementation language. That limitation can be removed by changing the language name argument to `null`, and making the feature implementation "free" (Figure 3.18).

```
manager.implementFeature(
    null,
    'entryPoint/any',
    ['entryPoint/any/sync'],
    async obj => ({
        entryPoint: async code =>
            obj.entryPoint.any.sync.entryPoint(code),
    })
);
```

Figure 3.18: *A free feature implementation*

invokes it immediately. In the less common situation where creating the entry point is asynchronous but using it is synchronous, the `requires/setup` feature from Section 3.2.3 may be of help, or the developer can simply create their own feature for that use case.

[13]`entryPoint/any` could still be specialized with `transformAsync` using the method described in the previous footnote, but there is no evidence of performance issues indicating that it would be worth the cost.

[14]This is the same for the plural version.

Free feature implementations (or "free features", analogous to "blanket implementations" in Rust) mean that languages only need to specify their most powerful features, and all strictly weaker features (that is, all features which can be implemented in terms of the more powerful features) will automatically be implemented. Equally powerful features can also be freely implemented in terms of each other, and Necode will safely handle them while avoiding cycles. Of course, even if a free feature is present, like in the case of `evaluate/any` for JavaScript, a more efficient language-specific implementation may still be used and will override any free implementations that may be available.

3.2.2.4 Feature Disjunction

Sometimes, there may be a weaker feature for which there are multiple more powerful features that could each independently implement it. For example, Necode has a built-in feature `repl/instanced` which lets an activity asynchronously initialize a read-eval-print-loop environment, and then asynchronously feed it inputs and receive outputs. This could be implemented in terms of `repl/instanced/startupSync`, which initializes synchronously but evaluates inputs asynchronously, or in terms of `repl/instanced/evalSync` which initializes asynchronously but can evaluate inputs synchronously.

No additional APIs are required to perform this kind of disjunctive implementation; the Core plugin just freely implements `repl/instanced` twice, once with each of the more synchronous variants. What is noteworthy about this technique is that it can be applied more broadly for any situation where there are multiple sets of features that could produce a desired effect. For example, say an activity wants to support evaluating code to produce any JavaScript object (`evaluate/any`), but it is willing to cope with just receiving strings (`evaluate/string`) if the target language cannot directly speak JavaScript. It could define a new feature `myPlugin/evaluate/anyOrString` and implement it like in Figure 3.19 to support both more and less powerful languages.[15]

3.2.3 `requires/` Features

Sometimes languages implementations will need additional help from Necode itself in order to work properly. For example, a language implementation which only supports a recent version of Chrome should not silently fail when opened in Firefox or Safari, it should be able to inform Necode so that Necode can warn users ahead of time and avoid confusion.

For this purpose, the `requires/` namespace is reserved. These features are not intended to be required by activities (and in fact cannot be), but are rather read by Necode to perform functionality *before* an activity with that language is loaded. Currently there are two `requires/`

[15]Implementation order is important here: note that `myPlugin/evaluate/anyOrString` is implemented using `evaluate/any` first. That means that the `evaluate/any`-based implementation is more likely to be preferred in the event of an inevitable conflict.

```
declare module '@necode-org/plugin-dev' {
    interface FeatureMap {
        'myPlugin/evaluate/anyOrString': {
            evaluateAny<T>(code: string): Promise<T>;
        } | {
            evaluateString(code: string): Promise<string>;
        };
    }
}

// ...

manager.implementFeature(
    null,
    'myPlugin/evaluate/anyOrString',
    ['evaluate/any'],
    async obj => ({ evaluateAny: obj.evaluate.any.evaluate })
);

manager.implementFeature(
    null,
    'myPlugin/evaluate/anyOrString',
    ['evaluate/string'],
    async obj => ({ evaluateString: obj.evaluate.string.evaluate })
);
```

Figure 3.19: *An example disjunctive feature implementation*

features which have been defined, as seen in Figure 3.20, though more could be added in the future without breaking existing languages.

```
interface FeatureMap {
    'requires/browser': {
        isCompatible(): boolean;
        getRecommendedBrowsers(): string[];
    };
    'requires/setup': {
        setup(): Promise<void>;
    };
    // ...
}
```

Figure 3.20: *All of the `requires/` features which have currently been defined*

- **`requires/browser`**– This feature allows languages to perform JavaScript feature detection before an activity loads through the `isCompatible` method, and provide alternative browsers that Necode can suggest to users using the `getRecommendedBrowsers` method.

- `requires/setup`– This feature allows languages to ask Necode to run asynchronous initialization code before an activity loads using the `setup` method.

3.3 RTC Policies

Necode's collaborative features are based on peer-to-peer connections using WebRTC. However, in order for those peer-to-peer connections to exist, there must be a centralized entity called a "signaling server" which establishes those connections. To decide which users to connect to each other, Necode's signaling server directs various user events (such as connecting and disconnecting) to modules called "RTC policies", which can programatically determine which links to establish.

3.3.1 Challenges

Unlike activities and languages, policies need to be run on the server and must persist state throughout the entire time an in-class activity is running. This presents various challenges for the application when attempting to allow third-party extensions. To enumerate them for future reference:

1. A poorly-optimized policy could bog down the entire signaling server infrastructure.
2. If the signaling server crashes or restarts, it could cause loss of policy state.
3. Policy bugs which result in invalid policy state can break the activity for everyone.
4. Running arbitrary code on your server, even if written by someone well-intentioned, can be dangerous.

From these challenges we can derive a number of constraints that our policy development API would have to follow. (2) tells us that we need some way of committing policy state to disk (e.g. in a database), which could work well with (3) to help roll back the policy state if some event causes an error. Further mitigating (3), automated testing tools could also help prevent bugs from occurring in the first place. To avoid the dangers of (4), we would also need to be able to sandbox policies, or at the very least prevent them from directly using I/O (including networking) and accessing the rest of the server state.

All of those are theoretically achievable while letting third-party developers write policies in plain JavaScript, despite some technical difficulties in setting it up. However, challenge (1) is much trickier to resolve. Timing out third-party code that takes too long to run could help keep performance issues limited to particular activity instances, but it would not resolve any underlying issues. The fundamental issue we encounter is that if we want policies to be able to have arbitrary behavior, we cannot ensure that they will finish running in a timely manner, or finish running at all per the halting problem.

However, while the halting problem applies to turing-complete languages, it does not necessarily apply to non-turing-complete languages. We believe that programmatic policies do not need to be capable of all computation in order to be useful in practice. Rather than attempting to create some special JavaScript environment to resolve all of our challenges, we made a brand new domain-specific programming language, MiKe, whose design helps us resolve all of our challenges at once.

3.3.2 MiKe Overview

This section contains a brief overview of MiKe and how it helps resolve the challenges we described in the previous section. For a more complete description, see Appendix A: MiKe.

MiKe is an event-based language which is intended to be invoked by an external host language (TypeScript in Necode's case). It cannot perform any I/O on its own (resolving (4)) and there is no "main" function, just `on` handlers as shown in Figure 3.21. MiKe's syntax is heavily inspired by languages like TypeScript and Zig and should be very readable to TypeScript programmers, which is good since everything else in a Necode plugin in written in TypeScript.

```
on join(user: User) {
    link(user, user);
}

on leave(user: User) {
    unlink(user, user);
}
```

Figure 3.21: *A simple policy demonstrating MiKe's* `on` *syntax*

Compared to TypeScript though, MiKe has very limited control flow. In fact, the `if` statement is the only control flow available, which immediately hints at how MiKe resolves issue (1) in the previous section. MiKe has no loops and thus only allows constant-time algorithms, which are very unlikely to cause performance bottlenecks and will not be negatively impacted by having a large number of users.[16] With that said, hosts are free to provide MiKe programs with *external* functions which take longer than constant time.

As part of mitigating issue (3), MiKe has no exceptions of any kind and every branch of an `if` statement must always be handled. Because we cannot throw null-reference exceptions, MiKe requires some safe mechanism to extract values from its `option<T>` type. Rather than opting for pattern matching which is less familiar in C-like languages, MiKe borrows from Zig's

[16]Note that in this context "constant-time" refers to O(1) time complexity. MiKe event handlers do not always take the same amount of wall time.

"payload capture" feature [26] and integrates value extraction directly into its `if` statement (Figure 3.22).

```
let maybeVal: option<int>;
// ...
if maybeVal |val| {
    // val: int
}
else {
    // maybeVal was none
}
```

Figure 3.22: *A demonstration of MiKe's if-deconstruction syntax*

To maintain persistent state between calls to event handlers, mutable global `state` variables may be declared with any serializable type (Figure 3.23). Serializable types include primitives and built-in non-primitive types (such as `option<T>` and `Map<T>`), host-defined types, and user-defined types whose fields only include other serializable types. While the particular implementation of serialization may vary depending on the target, the JavaScript target allows the entire state to be serialized as a JSON string and loaded back from a JSON string. This can be saved to a database and rolled back from in the event of some unexpected error, resolving (2).

```
state lastUserBox: option<User> = none;

on join(user: User) {
    if lastUserBox |lastUser| {
        if lastUser == user {
            debug user, "joined twice in a row!";
        }
    }
    lastUserBox = some(user);
}
```

Figure 3.23: *A demonstration of MiKe's `state` variables*

The last significant MiKe feature to be aware of is `param` variables. These are declared similarly to `state` variables but do nearly the opposite. `param` variables allow a MiKe program to tell the host that it needs to be parameterized by a set of values. The host is free to opt out of params if it cannot process them, but for Necode, params are essential for enabling third-party developers to have their activity configuration alter the behavior of their policies.

3.3.2.1 MiKe-Necode

While MiKe was designed with Necode in mind, the language is intentionally unaware of Necode's existence and is capable of supporting any host application which has similar requirements to that of Necode's signaling server. However, this means some glue is required in order for Mike and Necode to understand each other. The implementation of this glue is fairly mundane, but it is important to know which types, values, and events are made available to developers in order to understand the capabilities that Necode grants policy developers.

Necode-Provided Types

- `User`– A user. Has no fields, though may in the future (see Section 5.1.2.1).
- `Policy`– Represents a type of policy. May be used in `param` variables, and is intended to allow policies to create sub-groups with different policies (e.g. a breakout room policy could take an arbitrary policy to use for each room).
- `Group`– Represents a sub-group.
  - `.join: (User) => unit`– Adds a user to the group, mirroring the `join` event.
  - `.leave: (User) => unit`– Removes a user from the group, mirroring the `leave` event.
  - `.has: (User) => boolean`– Checks if a user is already in a group.
- `SignalData`– A wrapper around the data provided to a policy when signaling (Section 3.3.3).
  - `.getInt: (string) => option<int>`– Gets an integer field from the signal data, if it exists.
  - `.getFloat: (string) => option<float>`– Gets a float field from the signal data, if it exists.
  - `.getBoolean: (string) => option<boolean>`– Gets a boolean field from the signal data, if it exists.
  - `.getString: (string) => option<string>`– Gets a string field from the signal data, if it exists.

Necode-Provided Values

- `link: (User, User) => unit`– Links the first user to the second user. Note that links are directional, so a bi-directional link would require two calls.
- `unlink: (User, User) => unit`– Unlinks the first user from the second user. Unlinking users is also directional.
- `Group: (Policy) => Group`– The constructor of the `Group` type. Takes in a policy to use as the basis of the new group and returns the sub-group.

Necode's Events

- `join(User)`– Fired when a user joins.
- `leave(User)`– Fired when a user leaves.
- `signal(User, string, SignalData)`– Fired when a user sends a signal (Section 3.3.3). Contains the signal event and payload data in the second and third arguments respectively.

3.3.2.2 Tooling

Designing and implementing a new language is only the first step in making a usable programming language. In the modern day, IDE support for syntax highlighting, "red squiggles," code completion, and so on are a near-necessity. Fortunately, MiKe's query-based compiler architecture [27] allows for these kinds of tools to be built with relative ease, and so we have created a language server and Visual Studio Code extension for MiKe which provides all of the above and more.

However, out of the box MiKe knows nothing about Necode and what external types and functions it provides, leading to red squiggles on perfectly valid code when it encounters unrecognized variable and type names. To resolve this, we provide a mechanism for developers to put a `mike-config.ts` file in their workspace which will be automatically recognized by the language server and provide all the necessary configuration for local development.

3.3.3 Signals

A couple of times throughout this thesis we have mentioned a hypothetical policy involving "breakout rooms," but we have not detailed precisely what this would entail. Purely using the "join" and "leave" event handlers, it would be possible to write a policy in which users are automatically assigned into small groups when they join (where each small group is governed by a policy from a `param` variable). However, a common non-automatic way to split up students in applications like the tele-conferencing software Zoom is for an instructor to supply a set of pre-configured "breakout rooms" and allow students to join these rooms. This use case came up multiple times when discussing Necode's design with instructors at and outside of WPI, so it was important that Necode's policies support it.

The answer is signals. Signals are the third type of event that can be handled by Necode policies, and allow students in an activity to ask the policy to do something. Signals are fired on the client by retrieving a `signal` function from the `useSignal` hook and calling it with a signal event and payload object. Those are then passed on to the `signal` event handler described in the previous section, where the policy can choose to do whatever it wishes with it (such as add the user to or remove a user from a breakout room).

3.3.4 Automatic Validation

One of benefits of making MiKe is that its constraints mean policies can be automatically checked for compliance with certain rules without fear of impure effects getting in the way. In particular, we want to ensure the following invariants after firing an event:

- **Unlink after leave**– After a user leaves, they must be unlinked from all other users and they must have left all sub-groups.
- **Forget after leave**– After a user leaves, they must no longer be referenced in state.[17]

And we want to ensure that the following rules were followed *during* execution of an event handler:

- **No double-link**– For users `a` and `b`, if `link(a, b)` has been called, it must not be called again without `unlink(a, b)` being called in-between.
- **No unlink before link**– For users `a` and `b`, `unlink(a, b)` must not be called unless `link(a, b)` has been called at least once and after the most recent `unlink(a, b)` call.
- **No double-join**– For policy `p` and user `u`, if `p.join(u)` has been called, it must not be called again without `p.leave(u)` being called in-between.
- **No leave before join**– For policy `p` and user `u`, `p.leave(u)` must not be called unless `p.join(u)` has been called at least once and after the most recent `p.leave(u)` call.

To test for compliance with these constraints, we perform random testing using fast-check [28], which is based on the famous Haskell property-based testing library QuickCheck [29]. As a brief summary, fast-check allows developers to test whether a function complies with a certain property by using a randomized data generator called an "arbitrary", which can be constructed by hand, or using built-in arbitrary primitives and combinators. In our case, we can simulate using a policy by generating both values for the policy's params and an event sequence to run.

However, a naive attempt to do this will end up admitting clearly broken policies if they have more than a minimal amount of complexity. This is because the input space is so massive that it would require an infeasible number of iterations to test every scenario. For example, take the recurring example of breakout rooms. If each room is assigned an integer starting from 1, attempting to get random rooms by selecting random integer has such a large number of possibilities that, even with the biases that fast-check adds to better capture edge cases, it

[17]In an earlier design, policies were allowed to keep track of users even after they left, and there was a separate `forget(User)` event for if Necode wanted the policy to completely erase any memory of the user. This would have allowed policies to have different behavior for brief disconnections versus long-term disconnections. However, the design has since been simplified and determining what to do in the case of temporary disconnections is now the sole responsibility of Necode.

will likely never hit an event sequence in which a user tries to join the same room twice, even though that is a relatively likely scenario. If the policy had a bug where it did not remove a user from their existing room when they joined a new room, the validator would likely never find that the policy violated the “no double-link” rule.

3.3.4.1 Validator Configuration

We solved this issue by having policy developers declare constraints not just on the types of inputs, but also on which specific values (or ranges of values) should be permitted. At runtime, if an instructor attempts to start an activity with parameters that fall outside of these constraints or a user attempts to send a signal whose payload falls outside of these constraints, the request will be rejected outright before the MiKe event handler ever fires. This way, the validator only needs to generate the values which are considered acceptable by the configuration, reducing the number of possible inputs by many orders of magnitude.

This configuration is placed inside the plugin’s `necode.json` file (Section 3.4.2), within the the `config` field of a policy configuration object. In order to simplify development, the JSON-based validator configuration relies heavily on a `AtLeastOneOf<T>` utility type which allows developers to either put a single value or an array of possible values. For instance, if there are multiple signal events which both have the same constraints on their signal data, the event names can be put into an array to avoid duplication. The full `PolicyValidatorConfig` schema represented using TypeScript type definitions can be seen in Figure 3.24.

3.3.4.2 Building the Arbitrary

Even with the validator configuration, creating the arbitrary for our test input is non-trivial. The parameter side is relatively simple—for each parameter create an arbitrary which matches the configuration, and then use the `record()` combinator to combine them into a single object. However, the event sequence is more difficult to construct. We need to know how many users should we want to test on, we need to create join/leave/signal events in a reasonable proportion, and we need to ensure that the resulting event sequence is actually valid.

We do this in a multi-step process, shown in Figure 3.25. Starting with six users, labeled 0-5, we create two join and two leave events for each of them. If the policy has a signal handler, we also generate four random signal events for each user based on the validator configuration. It is possible that for more complex policies only six users or only four signal events will be insufficient, but for now we assume that these are sufficient to catch errors in most policies.

After gathering all of these events together, we use a built-in combinator to generate random permutations of the event sequence. However, these event sequences may not be valid. For example, just as policies must comply with the “no double-join” constraint, Necode must also avoid invoking the join event on the same user twice without them leaving in the middle. One solution to this issue could be to simply test if an event sequence is valid and filter it out if it is

```
type AtLeastOneOf<T> = T | [T, ...T[]];

type PolicyValidatorConfig = AtLeastOneOf<SingleValidatorConfig>;

interface SingleValidatorConfig {
    readonly params?: AtLeastOneOf<Values>;
    readonly signal?: AtLeastOneOf<SignalInfo>;
}

interface SignalInfo {
    type: AtLeastOneOf<string>;
    data: AtLeastOneOf<Values>;
}

type Values = { [name: string]: AtLeastOneOf<Value> };
type Value = NumberExact | NumberRange | String | Boolean;

interface NumberExact {
    type: 'int' | 'float';
    value: AtLeastOneOf<number>;
}

interface NumberRange {
    type: 'int' | 'float';
    ge: number;
    le: number;
}

interface String {
    type: 'string';
    value: AtLeastOneOf<string>;
}

interface Boolean {
    type: 'boolean';
    value?: boolean;
}
```

Figure 3.24: *The type definitions for policy validator configuration*

not. However, even just considering the join/leave events, there would only be a $(\frac{1}{6})^6 = \frac{1}{46656}$ chance that a random permutation would be valid.[18] When signal events are included that number rises exponentially, and even without them, generating thousands of inputs through this method would take far too long.

Instead, we attempt to "repair" invalid permutations by walking through them, and whenever we encounter an invalid event, simply discarding it from the sequence. This means that in practice most event sequences will have far fewer than the theoretical maximum of 48 events, but are still sufficiently long to provide a good sample of events, especially over thousands of iterations (see Appendix B: Expected Number of Events for more details).

However, the event sequences are not yet ready to use. When we encounter a duplicate test input, we would like to throw it away and get a new one so that each iteration is actually testing something new. However, some isomorphic inputs may have different data representations which result in the duplication not being properly detected. For example, consider the two complete event sequences:

1	join(user1)	leave(user1)
2	join(user4)	leave(user4)

These sequences test the exact same behavior, but will not be seen as duplicates because superficially, the user in the first sequence is labeled "user1" and the user in the second sequence is labeled "user4". The last stage of event processing is to fix this by normalizing the event sequences such that users are named in order of first appearance.

3.3.4.3 Branches

Some cases are known by the policy developer to be unreachable, for example if a signal event does not match any of the events specified in the validator configuration, or if a particular state is known to be impossible but that fact cannot be proven to the MiKe compiler. For these cases, Necode provides a `BUG` object which can be used with MiKe's `debug` statement (see Figure 3.26) to log an error and immediately bail on any event which hits it and revert to the previous state.

Any branch which does *not* contain a `debug BUG` statement should therefore be expected to be acceptable code for the policy to reach, and if the randomized testing does not reach that code, it is indicative of either a bug in the policy, a bug in the configuration, or an overly wide configuration that prevents the policy from being adequately validated. Thus, failing to reach a branch which does not have a `debug BUG` statement will also cause policy validation to fail with a message about the unvisited branch.

[18]For each user, the join (J) and leave (L) events must be arranged within the overall sequence in the order J-L-J-L in order for the configuration to be valid. The chance of this being the result of a random permutation for a single user is $\frac{1}{\binom{4}{2}} = \frac{1}{6}$ since we need to choose two of the four positions to be leave events. For six users, that is raised to the sixth power.

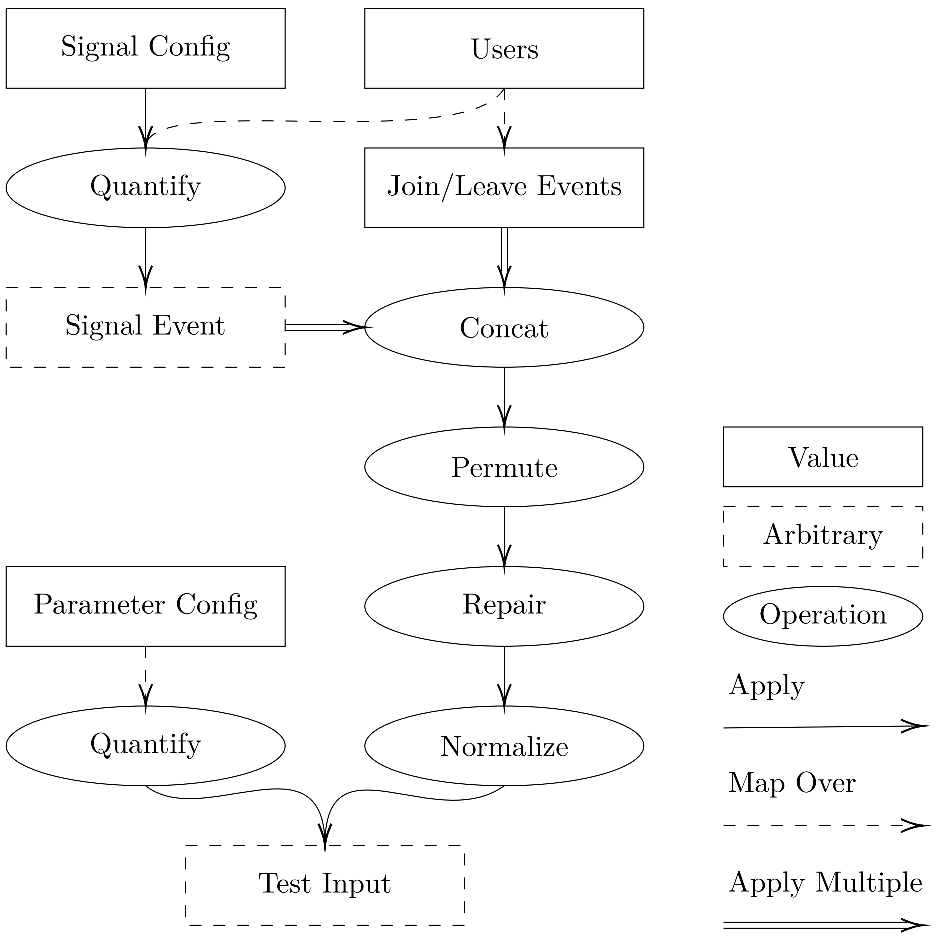


Figure 3.25: *A diagram of how an entire test input is packed into a fast-check arbitrary*

```
on signal(user: User, kind: string, data: SignalData) {
    if kind == "joinRoom" {
        // ...
    }
    else if kind == "leaveRoom" {
        // ...
    }
    else {
        debug BUG, "Unrecognized signal kind", kind;
    }
}
```

Figure 3.26: *An example of Necode's* ***debug BUG*** *pattern*

3.4 Plugins

In order to load activities, languages, and policies into Necode, a mechanism to make them available to Necode is required. One method could be static linking: modifying the Necode source code to add new modules before re-compiling and re-deploying it. However, while static linking is a perfectly fine experience for first-party developers, it becomes much trickier for third-party developers who want to offer their modules for institutions to install; static linking would require someone technical on staff to drop modules into the appropriate places in the Necode source, make sure the new modules are being statically referenced, and then re-deploy the new version.

The alternative to static linking is dynamic linking, where modules can be added to an instance of Necode without requiring the source to be modified or requiring Necode to be re-compiled. This is the approach Necode's plugin system takes. Even better, Necode's plugins allow adding and removing modules *hot*, meaning there is no downtime necessary when adding new features (though users may need to refresh the page).

In order to enable this, any combination of modules can be packed into a single plugin file and uploaded on the administrator plugin management page (Figure 3.27) where the list of currently-installed plugins can be viewed and, if necessary, removed. The precise mechanism for how these modules are bundled together will be discussed in the following sections.

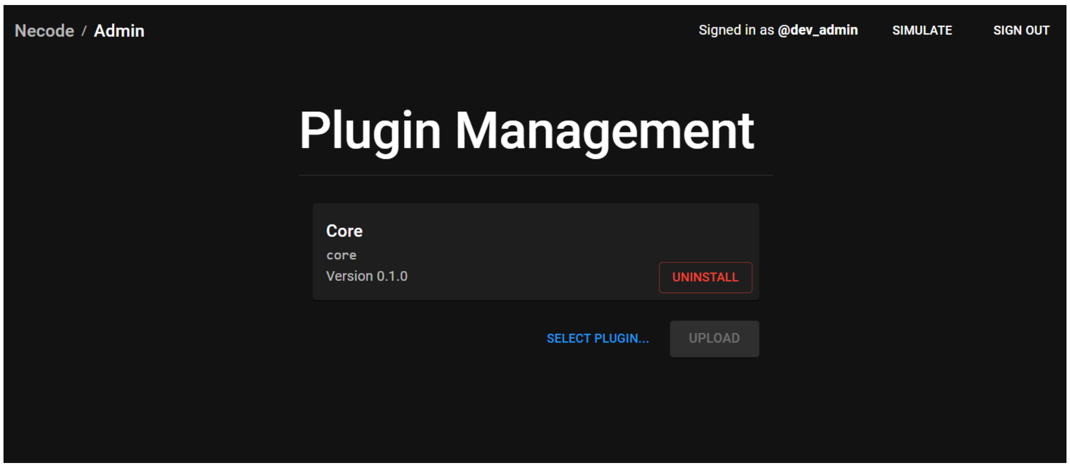


Figure 3.27: *A screenshot of the plugin management admin page*

3.4.1 Hooks

The entry point for all the front-end JavaScript, including activities and languages, is a file whose default export is a class extending the `Plugin` abstract class from `@necode-org/plugin-dev`. This class contains three optional methods called "hooks" which sub-classes may override to register their modules. These three hooks are `registerActivities`, `registerLanguages`, and `registerFeatures`.

The parameters to each hook method include a manager used to register a particular type of module. For example, `registerActivities` has one `ActivityManager` parameter which includes a `registerActivity` method to register an `ActivityDescription`. Section 3.2.2 describes how the `LanguageManager` and `FeatureManager` are used in their respective hooks and Figure 3.28 demonstrates a hypothetical plugin implementation.

This approach of having developers override optional hooks has a number of advantages over alternative approaches like having a single plugin function with an "everything manager" or having plugins export an object with arrays of activity and language descriptions. Our approach allows Necode to control when certain kinds of modules are loaded, for example to ensure that all available languages are known before Necode attempts to register any features. It also enables future expansion more easily than having an object of modules—new methods can be trivially added to existing managers without breaking existing plugins or adding significant complexity, and new hooks (either for new types of module or for other plugin events) can be introduced with no cost to developers who do not use them.

3.4.2 `necode.json`

A plugin's entry point cannot contain everything needed for, however. Since policies are only used by the signaling server, they are never referenced in any plugin hook. Additionally, for code splitting to work at all, there needs to be a way to tell Necode which files should be served to users. Plus there is some additional metadata which third-party developers will want to provide like the plugin name and version.

All of this information is packed into a JSON file in the root of a plugin called `necode.json`. This is where plugins are enumerated, plugin validator configuration (Section 3.3.4.1) is specified, and paths within a plugin's directory structure are declared. Additionally, though a plugin's `package.json` file is scraped to guess at fields like name, display name, and version, any of these fields may be overridden in `necode.json`'s `packageOverrides` field. An example of a `necode.json` file is provided in Figure 3.29. For convenience, we also provide developers with a JSON schema to get auto-completion and documentation while writing these configuration files.

3.4.3 Packing

In order to combine plugins into single files, Necode relies on existing infrastructure, treating plugins as ordinary NPM packages and having developers zip them up into a gzipped tar file with `npm pack`. When a plugin is uploaded, Necode will decompress and unzip them and extract the relevant files to the right places.

Bundling the front-end to prepare it for packing is a bit more complicated. Typically when a web application is served, it has been statically linked and bundled with all the libraries it uses, meaning dependencies are only loaded once. However, with dynamic linking, we can often

```
declare module '@necode-org/plugin-dev' {
    interface FeatureMap {
        'format/html': {
            toHtml(code: string): Promise<string>;
        };
        'format/dom': {
            toDom(code: string): Promise<HTMLElement>;
        };
    }
}

export default class MarkdownPlugin extends Plugin {
    override registerActivities(manager: ActivityManager): void {
        manager.registerActivity(markupPlayground);
    }

    override registerLanguages(manager: LanguageManager): void {
        manager.registerLanguage({
            name: 'markdown',
            monacoName: 'markdown',
            displayName: 'Markdown',
            icon: MarkdownIcon,
        });
    }

    override registerFeatures(manager: FeatureManager): void {
        manager.implementFeature(
            null,
            'format/html',
            ['format/dom'],
            async ({ format }) => ({
                toHtml: code => format.dom(code).then(x => x.outerHTML),
            }),
        );

        manager.implementFeature(
            'markdown',
            'format/dom',
            [],
            () => await import('./renderMarkdown')
                .then(m => ({ toDom: m.default }),
        );
    }
}
```

Figure 3.28: *An example plugin which adds support for writing with Markdown languages*

```
{
    "packageOverrides": {
        "name": "my-plugin",
        "displayName": "My First Plugin"
    },
    "frontendRoot": "dist",
    "entry": "index.js",
    "policyRoot": "policies",
    "policies": [
        {
            "path": "breakout.mike",
            "id": "my-plugin/breakout",
            "displayName": "Breakout Rooms",
            "config": {
                "params": {
                    "numGroups": { "type": "int", "ge": 2, "le": 50 }
                },
                "signal": [
                    {
                        "type": "joinRoom",
                        "data": {
                            "room": { "type": "int", "ge": 1, "le": 50 }
                        }
                    },
                    {
                        "type": "leaveRoom",
                        "data": {}
                    }
                ]
            }
        }
    ]
}
```

Figure 3.29: *A* `necode.json` *file based on Necode's core plugin's actual* `necode.json` *file*

into scenarios where both Necode and a plugin wish to use the same libraries and so they each have to use their own version of the library in their bundle.

In some cases that is fine and the only cost is that the user has to download the library twice, once when loading Necode and again when loading the plugin. But sometimes, especially when dealing with React context, things do not go so smoothly. React context works by first creating a “context” object, and then sharing that object between providers, which offer some value, and `useContext` hooks, which evaluate to the value stored in the nearest relevant provider. However, when the context provider is used in Necode and the context hook is used in plugins, they are actually referring to different context objects, since Necode and the plugin both have their own copy of the library and both create their own context object. For example, see how in Figure 3.30, both Necode and the plugin create different context objects from the same shared library and will be unable to communicate between them.

Necode bundle

```
// Shared library
export const MyContext = createContext();

// Necode code
export function MyComponent() {
    return <MyContext.Provider value={1}>
        <PluginContent />
    </MyContext.Provider>;
}
```

Plugin bundle

```
// Shared library
export const MyContext = createContext();

// Plugin code
export function MyComponent() {
    const ctxVal = useContext(MyContext);

    return <div>{ctxVal}</div>;
}
```

Figure 3.30: *An example of code using React context that causes issues with plugins*

The web bundling tool webpack provides a way out of this predicament. If Necode proactively exposes a known set of entry points to problematic libraries on the global object, webpack allows plugins to reference that global variable instead of bundling the library into the plugin. Necode currently exposes its own `@necode-org/plugin-dev` and `@necode-org/activity-dev`, MUI’s `@mui/system` so that developers can use the same design system as the rest of Necode, and `react` itself. As long as all plugins have the correct webpack configuration set up to reference these exposed libraries when building their front-end, everything should just work.

Fortunately, because the configuration has already been set up, the only thing third-party developers should need to do is run the `webpack` command with the provided webpack configuration.

3.4.3.1 Serving

How Necode serves front-end plugin code to users should be outside the scope of what third-party developers need to concern themselves with, but with custom webpack configurations, issues could arise. Necode takes advantage of a feature of webpack in which by default, files in a bundle will automatically detect the "public path," or the location of fellow bundle files, based on `import.meta.url` (the URL of the current module). This means that if a plugin entry point is served from a particular API route, say `/api/plugin/{pluginId}/files/index.js`, it will be able to automatically detect that its sibling files (from code splitting) should be located relative to `/api/plugin/{pluginId}/files`. This allows easy segregation of files between different plugins so long as plugin developers do not set an explicit public path in their webpack configuration.

Chapter 4

Evaluation

In “Writing Good Software Engineering Research Papers,” Shaw describes a number of possible strategies for evaluating work in software engineering papers [30]. One of these which was found to be effective was an “example”-based evaluation approach, in which the author creates demonstrations of their software in practice. Following that, to evaluate the effectiveness of our APIs and abstractions, we converted all of Necode’s existing built-in modules into a plugin and also created brand new activities, languages, and policies using our APIs from the start.

4.1 New Canvas Ring

Migrating the “canvas ring” activity (originally seen in Section 2.3.4) to the new APIs involved completely rewriting the portions related to RTC, which also happen to be the more interesting portions of the implementation (the rest is just typical React development). This consists of two parts: the ring policy itself and the communication between peers within the activity.

4.1.1 The Ring Policy

A ring can be implemented as a doubly-linked list which is connected at the ends, and that is how both the original implementation of the ring policy and the new MiKe implementation represent it. The MiKe implementation ends up being quite a bit smaller thanks to reduced boilerplate, but more importantly, it can leverage the benefits of automatic validation and serialization without requiring even more developer-written code. A simplified version[1] of the policy implementation can be seen in Figure 4.1.

[1]The full version is around 100 lines and can be found with the rest of Necode’s source code at https://github.com/TheUnlocked/Necode.

```
state ring: Map<User, RingItem> = {};
state head: option<RingItem> = none;

type RingItem (
    user: User,
    prev: User,
    next: User,
);

on join(user: User) {
    if head |headItem| {
        // 1+ users
        // ...
    }
    else {
        // ring is empty
        // ...
    }
}

on leave(user: User) {
    if head |headItem| {
        if ring.get(user) |item| {
            if item.user == item.next {
                // One user
                // ...
            }
            else {
                // 2+ users
                // ...
            }
            // ...
        }
        else {
            debug BUG, user, "left even though they weren't in the ring";
        }
    }
    else {
        debug BUG, user, "left even though the ring was supposedly empty";
    }
}
```

Figure 4.1: *A simplified version of the new ring policy written in MiKe*

To compare the performance of the MiKe policy with the old hand-written policy, we used the Benchmark.js library [31] which returns results in ops/s. We performed three benchmarks,[2] one in which one user joined and then left, one in which 100 users joined and then left, and one benchmark for instantiating the policy.[3] Surprisingly, we found that MiKe *beat* the performance of the hand-written code by a factor of 3-5x when handling join/leave events, though its performance suffered greatly in instantiation with a 99.5% performance penalty (Figure 4.2). The MiKe policy could still be instantiated tens of thousands of times per second though, so that is unlikely to be a significant concern.

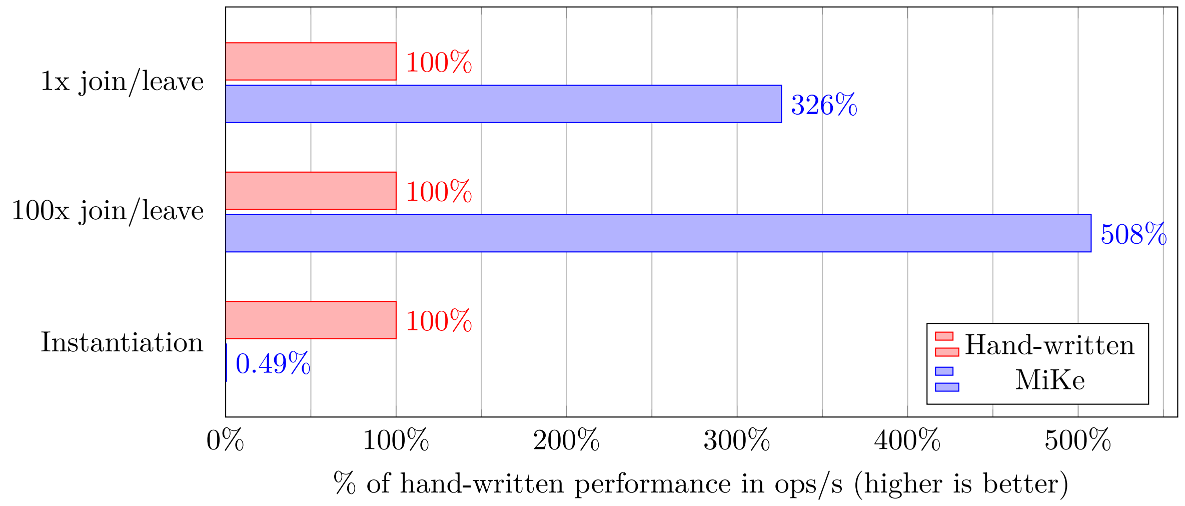


Figure 4.2: *Performance comparison of the old and new ring policy*

We suspect that the performance gains are due to the original policy focusing on readability over performance, while the emitted JavaScript after compiling the MiKe program is not concerned with aesthetics. This data should not be construed to imply that MiKe has better performance than hand-written code could reasonably achieve, just that MiKe policies are sufficiently fast for this application.

4.1.2 Using Media Channels

Though Necode prior to this thesis did create peers for activities, sending data through peers was entirely handled by activities themselves. The canvas ring activity had 70-80 lines of code[4] ensuring that media streams were correctly sent and handling issues that would inevitably arise when a stream arrived at the wrong time or video stopped being sent. With the new

[2]All benchmarks were run on a laptop connected to wall power with an Intel Core i7-8750H mobile CPU boosting to approx. 3.9 GHz.

[3]These actions are being benchmarked by directly invoking them. Networking/request-handling overhead is not included, and linking/unlinking is a no-op.

[4]The number varies depending on how you count. Lines of code or number of statements are certainly not the best way to measure complexity, but they do effectively convey the idea that `useMediaChannel` makes sending media streams over RTC much simpler to do properly.

`useMediaChannel` hook (Section 3.1.3.1), the activity has been substantially simplified, taking just one statement to establish the communication and only a few more to make sure that the video is coming in and the canvas is getting sent (Figure 4.3).

```
export function CanvasActivity(/* ... */) {
    const [canvas, setCanvas] = useState<HTMLCanvasElement | null>(null);
    const [[inboundStream], setOutboundStream] =
        useMediaChannel(NetworkId.NET_0, 'canvas');

    // ...

    useEffect(() => {
        if (canvas) {
            setOutboundStream(canvas.captureStream(FRAME_RATE));
        }
    }, [canvas, setOutboundStream]);

    const onCanvasRefChange = useCallback((canvas: HTMLCanvasElement) => {
        // ...
        setCanvas(canvas);
    }, []);

    return /* ... */
        <DrawingCanvas ... ref={onCanvasRefChange} />
        <Video ... srcObject={inboundStream} />
    /* ... */;
}
```

Figure 4.3: *The canvas ring activity implementation, truncated to show only RTC-related code*

4.2 Shared Editors

Integrating shared documents into the HtmlTestActivity base was relatively straight-forward, with some caveats. Creating shared text with `useY` and `useYText` and passing that into an `<Editor>` element was fairly simple. Adding awareness with `useYAwareness` was as well. However, other code which relied on the previous `useReducer`-based[5] state model to provide functionality unrelated to the editor required some more involved tweaking to make it work with Yjs documents. The reactive handles did help with that, but it may be a sign that developers should favor using `useY`/`useYText` for user input even if the activity doesn't support RTC,[6] so that their code will be conducive to adding shared editor support in the future.

[5]The `useReducer` hook comes from React, and is like `useState` but with some additional functionality.

[6]Necode provides `NetworkId.OFFLINE` which can be used as the network ID for the various RTC hooks to help with this.

Because Yjs operates over peer-to-peer connections and does not have a central server tracking document state, establishing *initial* document state (e.g. starter code) is not quite as simple as it is when state is purely local. In Section 3.1.3.3 we briefly mentioned the `useYInit` hook, which can be used to establish shared initial state in a safe manner. This hook is not particularly hard to use (Figure 4.4), but it does involve interacting with Yjs directly. Seeing as we had already worked with Yjs when integrating it into Necode, that was fine for us, but it is possible that `useYInit` would be somewhat jarring for someone who otherwise exclusively controls the Yjs document using the `Editor` component.

```
const y = useY(network, 'shared-editors');

useYInit(y, doc => {
    for (const type of ['html', 'css', 'code'] as const) {
        const initialValue = ...;
        if (initialValue) {
            doc.getText(type).insert(0, initialValue);
        }
        setCommittedState(st => ...);
    }
});
```

Figure 4.4: *A simplified version of how useYInit is used in the HtmlTestActivity source*

Yjs only requires a strongly connected (not complete) network, so the uni-directional ring policy is perfectly capable of synchronizing document state between users, something which we have demonstrated in toy scenarios where every client exists on the same local machine. However, relying on only uni-directional communication may cause unwanted synchronization latency for a large number of users, especially if they are on different networks. Due to the challenges involved in simulating a representative scenario, we unfortunately could not conduct measurements of that in this thesis, though a policy which is designed to establish more than one link, or even just a bi-directional version of the ring policy, could help if latency is discovered to be an issue.

4.3 Breakout Rooms

Multiple instructors we have talked to have mentioned that being able to have students break apart into small groups—often called breakout rooms—would be an important use case if they were to integrate Necode into their class, and breakout rooms have been used throughout this thesis as an example of something Necode's new third-party developer APIs should allow someone to implement. Though perhaps unsurprising given that the APIs were designed with support for this feature in mind, we successfully used our APIs to implement both backend and frontend support for splitting students into breakout rooms.

4.3.1 Policy

Breakout rooms, as we have implemented them, allow students to signal the policy about which particular room they would like to join. The number of rooms and the behavior of networking within each room are parameterized and can be configured by the activity, meaning the breakout policy itself is able to avoid linking users entirely and delegate that to another policy, for example the ring policy. Figure 4.5, a truncated version of the breakout policy, demonstrates how we were able to combine signals, params, and sub-groups in a MiKe policy to achieve that.

4.3.2 User Interface

Because the breakout policy can be parameterized by any other policy, we wanted the breakout room user interface to similarly be able to parameterize its activity. Thanks to the composability of React and the flexibility of our hooks, we were able to implement a `BreakoutRoomProvider` component entirely in plugin code which can wrap around existing activities to add a breakout room selector strip which correctly handles signaling on its own (Figure 4.6).

Naively keeping the exact same front-end code while adding the ability to switch breakout rooms causes synchronization issues with Yjs, since Yjs has no knowledge that different rooms are supposed to have different shared documents. To get around this, `BreakoutRoomProvider` provides the current room number to its children (i.e. the wrapped activity), which can then use it as part of the channel name in the `useY` hook (Figure 4.7). This design does require that the activity is to some degree “in on it” when adding breakout room support, but because this is implemented entirely by a plugin, other third-party developers could experiment with creating their own breakout room abstractions if this one does not fit their needs.

One potential pitfall we encountered is that because all program state is stored in the clients and not on the server, if a student leaves their room and joins another, the code in their original room will be lost (unless another student stayed behind in that room to preserve it). We would need to conduct user studies to understand how much of an issue that would end up being in the real world.[7]

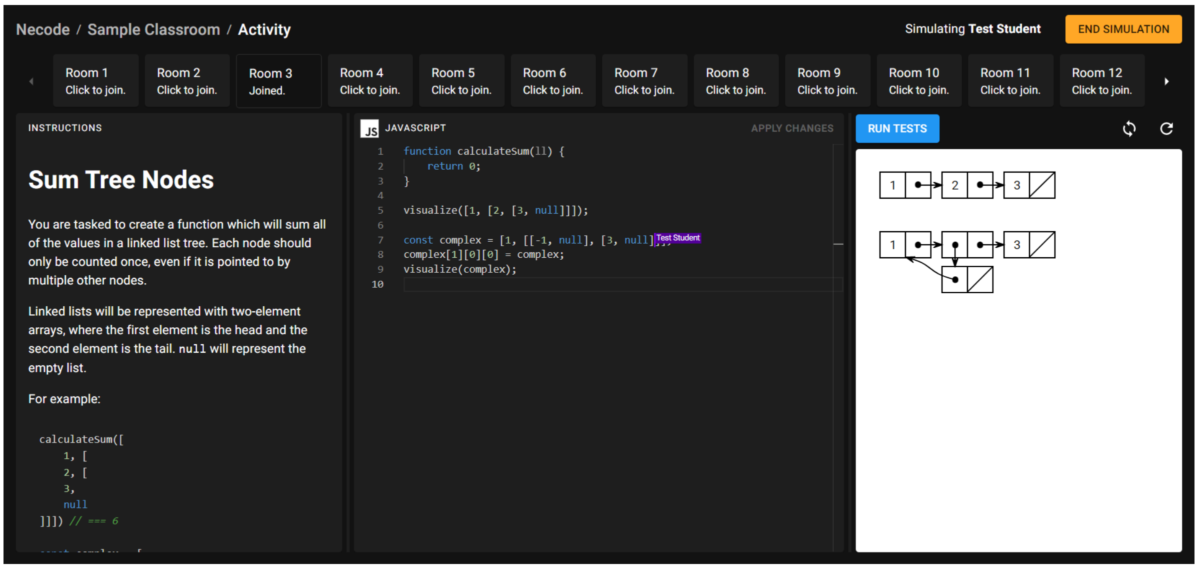
4.3.2.1 Widget

Sometimes groups have names rather than just numbers, and our breakout room proof of concept utilizes our widget components (Section 3.1.5.2) to allow instructors to configure both room names and the number of breakout rooms. The “Edit Rooms” button in Figure 4.8 will open a dialog that contains a plain text code editor where the instructor can put each room name on its own line.

[7]An alternative solution for breakout rooms would be to have everyone in the same group (e.g. with the ring policy) using different text fields on a shared document for each room. However, that would mean every student would be transmitting the code for every breakout room and could potentially have synchronization latency issues.

```
param numGroups: int;
param roomPolicy: Policy;

state groups: Map<int, Group> = {};
state groupMembership: Map<User, Group> = {};

on signal(user: User, kind: string, data: SignalData) {
    if kind == "joinRoom" {
        if data.getInt("room") |room| {
            if room > 0 && room <= numGroups {
                // Leave current group, if applicable
                // ...

                // Join existing group or create a new group
                if groups.get(room) |group| {
                    group.join(user);
                    groupMembership.set(user, group);
                }
                else {
                    let group = Group(roomPolicy);
                    groups.set(room, group);
                    group.join(user);
                    groupMembership.set(user, group);
                }
            }
        }
        else {
            debug BUG, "Signal data missing room field";
        }
    }
    else if kind == "leaveRoom" {
        if groupMembership.get(user) |group| {
            group.leave(user);
            groupMembership.remove(user);
        }
    }
    else {
        debug BUG, "Unrecognized signal kind", kind;
    }
}

on leave(user: User) {
    // ...
}
```

Figure 4.5: *A simplified version of the breakout policy written in MiKe*



Sum Tree Nodes

You are tasked to create a function which will sum all of the values in a linked list tree. Each node should only be counted once, even if it is pointed to by multiple other nodes.

Linked lists will be represented with two-element arrays, where the first element is the head and the second element is the tail. `null` will represent the empty list.

For example:

```
calculateSum([
    1, [
    2, [
    3,
    null
]]]) // === 6
```

```
function calculateSum(ll) {
    return 0;
}

visualize([1, [2, [3, null]]]);

const complex = [1, [[-1, null], [3, null]]];
complex[1][0][0] = complex;
visualize(complex);

```

Figure 4.6: *The breakout room user interface we created, wrapped around a DOM Programming activity*

Wrapped Activity

```
function Activity(props: ...) {
    const { rooms } = props.activityConfig;
    return <BreakoutRoomProvider
        network={NetworkId.NET_0}
        numRooms={rooms.length}
        roomNames={rooms}
    >
        {roomId => <InternalActivity roomId={roomId} {...props} />}
    </BreakoutRoomProvider>;
}
```

Original Activity

```
function Activity(props: ...) {
    const {
        roomId = ''
        ...
    } = props;
    // ...
    const y = useY(network, `shared-editors-${roomId}`);
    // ...
}
```

Figure 4.7: *An example of how an activity could be wrapped in a* `BreakoutRoomProvider`

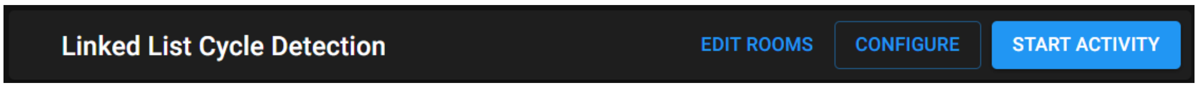


Figure 4.8: *The widget for the breakout room version of the DOM Programming activity*

4.4 Scheme + REPL

To demonstrate the potential of our language/feature and activity APIs, we implemented support for R6RS Scheme and a small REPL-based activity. Scheme REPLs (e.g. in DrRacket) are often used in computer science classrooms, and by integrating them into Necode, it opens the door for Necode to be used as a platform for social/collaborative activities involving Scheme.

4.4.1 Scheme in the Browser

Racket's fork of Chez Scheme (R6RS) is open-source and includes a WebAssembly compile target which allows it to be used in browsers. Some additional work is required to make this interface well with JavaScript, though that work is largely unrelated to this thesis. In short, we run the WASM-compiled Chez Scheme in a web worker, utilizing a `SharedArrayBuffer` with JavaScript's `Atomics` API to transmit stdin to the interpreter so that we can interface with POSIX's blocking I/O.[8] The full implementation of our `chez-scheme-js` library can be found at https://github.com/TheUnlocked/chez-scheme-js.

More of concern for this thesis is that because of browser mitigation for the Spectre vulnerability [32], `SharedArrayBuffers` require pages to be cross-origin isolated by setting the COOP (Cross-Origin Opener Policy) and COEP (Cross-Origin Embedder Policy) headers [33]. Necode provides these headers, but as to not break potential activities which may want to embed third-party content, Necode sets COEP to `credentialless`, a value which is currently only supported in Chromium browsers. Thus, our Scheme implementation only supports Chromium at the moment. In order to ensure activities using Scheme do not silently break, we implemented the `requires/browser` feature (Section 3.2.3) for Scheme with `SharedArrayBuffer` feature detection (Figure 4.9). This ensures that browsers which do not support `SharedArrayBuffer` (whether because they do not support `credentialless` or for some other reason) will display an error message to users before the activity loads.

4.4.2 Abstract REPL

In order to make a REPL activity, we first needed to define a feature that would allow the activity to invoke the REPL. We created several of these with different levels of synchronicity and gave the asynchronous features free implementations based on more synchronous variants, like was discussed for the `evaluate/` namespace in Section 3.2.2.3. Ultimately, to enable

[8]Blocking I/O does not work very well in a browser, where we want to be able to perform UI and networking tasks while the user is deciding what to input.

```
"requires/browser": {
    isCompatible() {
        return typeof SharedArrayBuffer !== 'undefined';
    },
    getRecommendedBrowsers() {
        return ['Chrome 96+', 'Edge 96+'];
    },
}
```

Figure 4.9: *The `requires/browser` implementation for R6RS Scheme*

maximum language coverage, our new REPL activity depends on `repl/instanced`, the most asynchronous variant in which a REPL environment can be instantiated asynchronously and queried asynchronously (Figure 4.10).

```
'repl/instanced': {
    createInstance(): Promise<{
        evaluate(code: string): Promise<ReplResult>;
        destroy?(): void;
    }>;
};
```

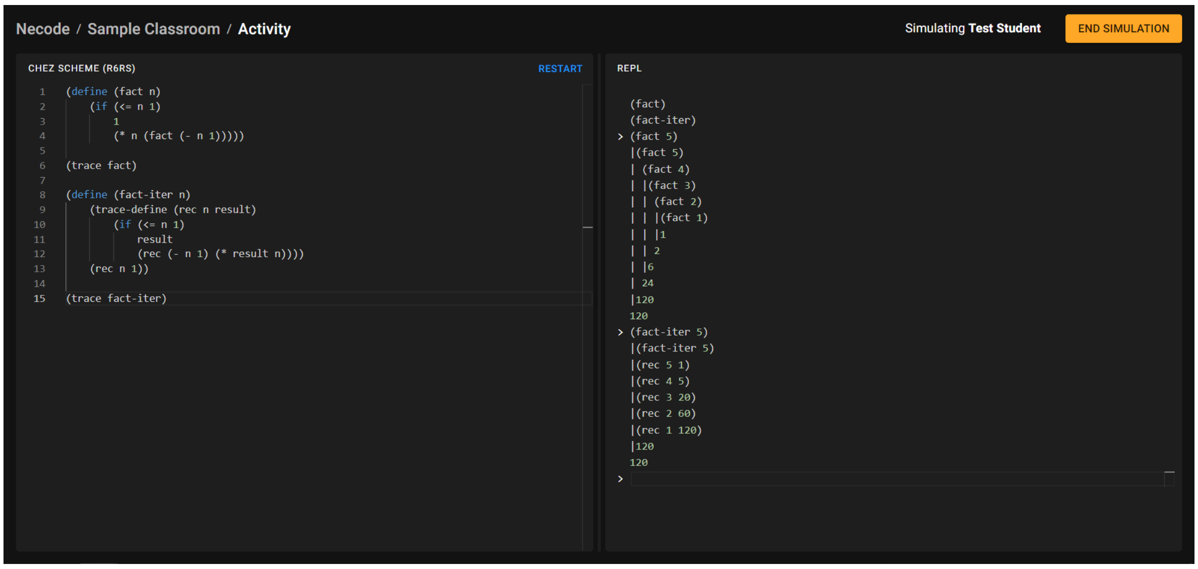
Figure 4.10: *The `repl/instanced` feature definition*

From there we created the REPL activity (Figure 4.11), which presents a definitions pane side-by-side with a REPL pane, similar to DrRacket's interface. Aside from sharing the same visual style,[9] this REPL activity is completely different from both the HtmlTestActivity family of activities and the canvas ring, demonstrating that our APIs provide the ability to make a broad range of user interfaces, not simply slight variations on built-in defaults. Additionally, the new REPL activity can still leverage the same features that other activities can. For example, because other languages also implement `repl/instanced` (directly or indirectly), the REPL activity can be used with not just R6RS Scheme, but also with JavaScript, TypeScript, and Python 3. Further, though the REPL activity currently does not use any form of collaboration or other real-time communication, it easily could using the same RTC and submission hooks as HtmlTestActivity uses and even using the same breakout room infrastructure we discussed in the previous section.

4.5 Size Overhead

After migrating everything to our new APIs and creating new modules, we packaged the core plugin and successfully dynamically loaded that plugin into Necode without any part of it being statically linked. With that said, in our live production environment, we still statically

[9]Thanks to the shared UI library and our `Panes` component.



```scheme
(define (fact n)
    (if (<= n 1)
        1
        (* n (fact (- n 1)))))

(trace fact)

(define (fact-iter n)
    (trace-define (rec n result)
        (if (<= n 1)
            result
            (rec (- n 1) (* result n))))
    (rec n 1))

(trace fact-iter)
```

```
  (fact)
  (fact-iter)
> (fact 5)
  |(fact 5)
  | (fact 4)
  | |(fact 3)
  | | (fact 2)
  | | |(fact 1)
  | | |1
  | | 2
  | |6
  | 24
  |120
  120
> (fact-iter 5)
  |(fact-iter 5)
  |(rec 5 1)
  |(rec 4 5)
  |(rec 3 20)
  |(rec 2 60)
  |(rec 1 120)
  |120
  120
>
```

Figure 4.11: *A screenshot of R6RS Scheme being used in a REPL-based activity*

link the core plugin due to increased overhead from dynamic linking. Using a student account in Edge with no extensions and caching disabled, we found that dynamic linking inflicted approximately 19% overhead of transferred data in JavaScript requests for pages which did not load any activities or languages, and approximately 9% overhead for an unconfigured "DOM Programming" activity configured for students to write in JavaScript (Figure 4.12). The overhead remained a relatively constant 146-147 KB in all pages which is why percentage overhead is smaller in a live activity (more more overall JavaScript is being transferred), but since the core plugin will be included in all Necode instances anyways, accepting even a smaller relative overhead is unnecessary.

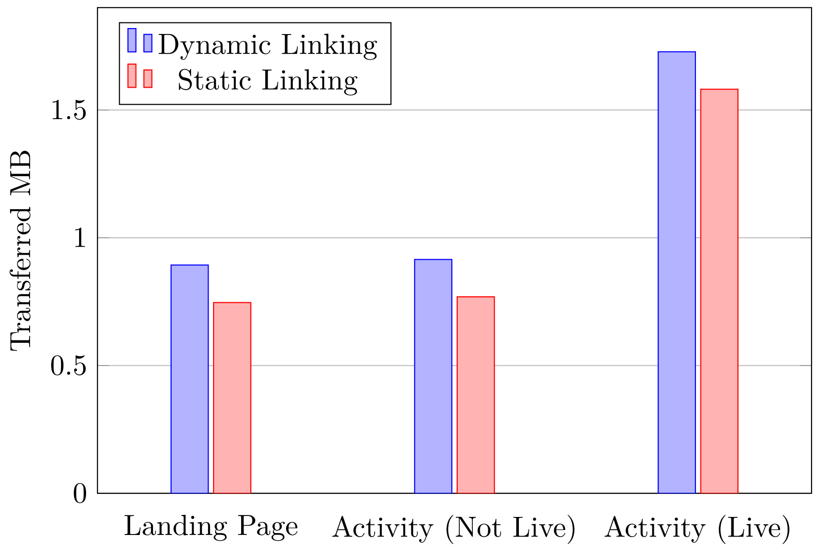


Figure 4.12: *Data transferred fetching JS when dynamically vs statically linking the core plugin*

Chapter 5

Conclusion

In this thesis we made substantial progress in designing effective APIs and abstractions to allow third-party developers to extend educational programming environments with their own activities, languages, and social/collaborative features. While we focused on specifically extending Necode, the concepts we developed should be applicable beyond it, and we hope that the work we have done in this thesis will help improve the state of third-party development tools for other environments in the wild.

5.1 Future Work

While we made significant progress in developing APIs for third-party developers, this thesis is merely one iteration which could be refined many more times in the future. In fact, we have some of our own ideas for directions that this work could branch into. In this section, we discuss some of the areas that we believe could benefit from further exploration in future works relating to Necode's third-party development APIs.

5.1.1 Languages and Features

5.1.1.1 Feature Specificity

Currently when there are multiple available implementations for a feature, predicting which will be used can be challenging. The current rules generally favor language-specific implementations over free implementations and earlier implementations over later ones, though this cannot be relied on in all cases. Usually which implementation is chosen would not matter because all implementations should have the same behavior, but with disjunctive features (Section 3.2.2.4), a developer may prefer that one implementation is chosen over another. If this becomes an issue in practice, concrete rules for declaring feature implementation specificity may be warranted.

5.1.1.2 Custom Editors

Because code editors in activities will usually use our own `Editor` component, the editor may be able to be replaced by something other than Monaco. For example, on mobile, we may be able to use CodeMirror 6 instead, which supports touch interaction [34].

However, for visual programming languages, we may be able to go even further than that. For example, it may be possible for a hypothetical `requires/editor` feature to allow a custom editor to replace the built-in editor so that visual programming languages can provide their own mechanism of writing code. It is unclear if such a design would be useful in practice though, so further exploration is needed.

5.1.1.3 `requires/` Feature Improvements

While we believe that the design of `requires/` features will be sufficient in most cases, there are three main scenarios in which they could fall short:

1. A developer wants to extend a language added by another plugin, but a `requires/` feature is necessary to do that and the language already has an implementation for that `requires/` feature.

2. A free feature needs a `requires/` feature.

3. A `requires/` feature only applies to a subset of the feature implementations for a language, and does not need to be obeyed when those are not used.

Of the three, we believe scenario (3) is the most likely, but it is also the least likely to cause harm. Scenario (3) should not cause anything to break, it simply might cause an unnecessarily degraded experience. Overall, we are somewhat skeptical that any of these scenarios are likely to cause issues in practice, but if they do, additional design work in this space may be required.

5.1.2 Policies

5.1.2.1 User Roles

The `User` type Necode exposes in MiKe policies currently has no fields, leaving policies completely blind as to the identity of the users they are linking. However, some policies may want to be able to treat students and instructors differently. Exposing a “role” field to policies could enable more interesting policies, and we would like to explore this in the future.

5.1.2.2 Constrained Polynomial-Time Policies

Currently, due to MiKe not supporting iteration or recursion, policies are significantly constrained in what they can do. For example, a policy could not currently create a fully connected network, since doing so would require iterating over every other user in the network. Keeping control flow limited to avoid dangerous performance pitfalls is valuable, but allowing some constrained low polynominal-time control flow would be valuable. We would like to explore the potential ways of doing this in the future.

5.1.2.3 Parameterized Validator Configuration

Validator configuration can currently only take static values and value ranges when configuring the domain of params and signal payloads. That can create gaps, such as how in the breakout rooms policy, the validator configuration cannot change the allowable range of the signal payload's `room` field based on how many rooms there are in the `numGroups` param. Allowing the constraints of fields to depend on other fields could be valuable, though care would need to be taken to ensure it does not cause issues with the random testing validation algorithm.

5.1.3 Plugins

Currently, in order for a component in core to be used by other plugins (e.g. the `BreakoutRoomProvider` component from Section 4.3.2), it would need to be published as a separate NPM library and then included in the build of other plugins, causing duplicated code to be downloaded by users. In order to fix this, it may be possible to create a mechanism by which plugins can export objects to be used by other plugins. Allowing this would not only require designing an export/import system, but also a plugin dependency system and a way to ensure that imports are well-typed for use in TypeScript. Despite the challenges involved, we are interested in exploring the possibilities of this in the future.

5.2 Final Notes

Necode is open for use by WPI faculty at https://www.necode.org/,[1] and instructors at other universities (or contributors anywhere) can find the complete source at https://github.com /TheUnlocked/Necode, along with some documentation on setting up their development and production environment.[2]

[1]Or https://code.cs.wpi.edu/

[2]Production environments being run at other universities may require some alterations to how authentication works to hook into their SSO. Remember to publish any modifications to comply with the AGPL 3.0 license, or even better, make a pull request!

Appendix A: MiKe

This appendix will describe MiKe's design. However, it will largely not cover rationale for the design decisions, which are discussed in the context of Necode in the main body of this thesis.

MiKe is a language for defining event handlers. These handlers are invoked by some "host," typically a JavaScript program,[3] which can provide its own types, functions, and other values for the handlers to use. In addition to defining event handlers, MiKe programs can declare state variables which are used to persist data between events, and the host may allow programs to request parameters of certain types. In the JavaScript code generation target, serialization code is automatically generated for all state variables, which enables an entire program state to be saved and restored at a later point.

Top-Level Statements

There are four types of top-level statements in MiKe, whose approximate syntax in Antlr4 notation [35] are shown below:

```
topLevelStatement
    : paramDeclaration
    | stateDeclaration
    | eventHandler
    | typeDefinition
    ;

paramDeclaration: 'param' NAME ':' type ';';
stateDeclaration: 'state' NAME (':' type)? '=' expression ';';
eventHandler: 'on' NAME '(' parameterList? ')' block;
typeDefinition: 'type' NAME parameterList ';';

parameterList: NAME ':' type (',' NAME ':' type)*;

type
```

[3]JavaScript is currently the only target language/format for which code generation has been implemented.

```
    : NAME ('<' (type (',' type)*)? '>')?   // X, X<T, ...>
    | '(' (type (',' type)*)? ')' '=>' type // functions
    ;

block: '{' statement* '}';
```

Param Declaration

```
param x: int;
```

Param declarations are used to notify the host that the program would like to be parameterized by a value of some type, and create an unassignable param variable in the top-level scope that can be referenced by other expressions. Because the host must understand these declarations, the types that can be used for param variables are highly limited. In order for a type to be accepted it must have the `[IsLegalParameter]` attribute, which is present on `boolean`, `int`, `float`, `string`, `option<T>` (if `T` has the attribute), and any host-defined type which includes the attribute.

State Declaration

```
state x: int = 0;
state x = 0;
```

State declarations create a state variable in the top-level scope which may be referenced, reassigned, or mutated in event handlers. Code generation targets are expected to provide a way for the host to serialize and deserialize all state variables. Because functions cannot be easily serialized, they are not legal types for state variables, and neither are any user-defined types which include function type fields, nor generic types which are parameterized by other types which would be illegal to use in state declarations.

An expression must always be provided to state variables in order to indicate the initial value of the state. If an explicit type is provided, it will be used to target type the right-hand side expression. Otherwise, the type of the state variable will be inferred from the type of the right-hand side expression, if possible.

Event Handler

```
on eventName(arg1: int, arg2: string) {
    // ...
}
```

Event handlers take an argument list with types pre-defined by the host so that the host can correctly invoke the event handlers. Within an event handler, any statements may be used to perform effects, typically either altering program state, invoking functions provided by the host, or both.

Type Definition

```
type MyType(field1: int, field2: string);
```

Type definitions define a type with a set of mutable fields, and a constructor function with the same name as the type and whose parameters follow the field list in the type definition. All types defined in this manner have the `[IsUserDefined]` attribute.

Statements

There are seven types of statement in MiKe, whose syntax is shown below:

```
statement
    : expression ';'
    | variableDefinition
    | variableAssignment
    | dereferencingAssignment
    | ifStatement
    | debugStatement
    | block
    ;

variableDefinition
    : 'let' NAME '=' expression ';'
    | 'let' NAME ':' type ('=' expression)? ';'
    ;

variableAssignment: NAME '=' expression ';';

dereferencingAssignment: derefOrApply '.' NAME '=' expression ';';

ifStatement: 'if' ifCase ('else' 'if' ifCase)* ('else' block)?;
ifCase: expression ('|' NAME '|')? block;

debugStatement: 'debug' expression (',' expression)* ';';
```

Variable Definition

```
let x: int = 0;
let y = 0;
let z: int;
```

Variable definitions declare a local variable in the current block scope from the location of the statement onward. Local variables may be shadowed, but not within the same block scope. If a type is explicitly given, the right-hand side will be target typed to the given type, otherwise the type of the variable will be inferred from the right-hand side expression if possible. If no expression is provided, an explicit type must be given, and the variable will be considered “uninitialized” and thus unusable in locations where it is not definitely assigned (i.e. where it cannot be proven on a syntactic level that the variable will have been assigned there).

Variable Assignment

```
x = 1;
```

Variable assignment changes the value of a local or state variable to the result of the right-hand side expression, which is target-typed by the type of the left-hand side variable.

Dereferencing Assignment

```
x.field = 1;
y.foo().field = 2;
```

Dereferencing assignment allows a field on a type with the `[IsUserDefined]` attribute to be mutated into the result of the right-hand side expression, which is target-typed by the type of the left-hand side dereferencing expression.

If Statement

```
if condition {
    // ...
}
else if condition2 {
    // ...
}
else {
    // ...
}
```

```
if opt |x| {
    // ...
}
```

The type of the condition expression in an if statement must have the `[IsLegalCondition]` attribute, which is only present on `boolean` and `option<T>` by default, though the host may include it on other host-provided types. If the result of the condition expression is a “truthy” value, the block of that case will be entered. Otherwise, if there is an `else` block, that block will be entered. The `else if` construct is equivalent to an if statement inside of an `else` block with chained `else if` and `else` clauses being bound in a right-associative manner.

For `booleans`, `true` is truthy and `false` is not. For `option<T>`, a value is truthy if and only if its `hasValue` field is `true`. For host-defined types, truthiness is target-defined.

If the condition type’s `[IsLegalCondition]` attribute has the `destructInto` field set, the destructuring syntax may be used to extract a value from the condition and bind it to a variable whose type is that of the `destructInto` field within the associated block’s scope. For `option<T>`, that type is `T` and the extracted value is the value stored in the option. For host-defined types, `destructInto` may be defined by the host and the extracted value is target-defined.

Debug Statement

```
debug a;
debug a, b, c;
```

Debug statements evaluate their input expressions and send the resulting values to the host to process. Debug statements should typically not alter the behavior of the event handler, though that is ultimately up to the discretion of the host.

Expressions

Expressions are broken up into “operator” expressions which do not fully enclose constituent expressions, and “contained” expressions which are either atoms (i.e. they have no constituent expressions) or are enclosed by syntactic features on both sides.

Operator Expressions

MiKe’s operator expressions include thirteen binary operators, two unary operators, and function application. These operators are left-associative and exist at six levels of precedence. In ascending order, these are:

1. `&&`, `||` The logical AND and OR operators. These take in two `booleans` and return a `boolean`.

2. `==`, `!=`, `<`, `>`, `<=`, `>=` The relational operators. `==` and `!=` take in two expressions of any type[4] and check if their values are equal/not equal. The other operators take in two `int` or `float` values and perform their respective comparisons, returning a `boolean`.

3. `+`, `-` The add and subtract operators. These take in either two `int`s or two `float`s and return the same type.

4. `*`, `/` The multiply and divide operators. These take in either two `int`s or two `float`s and return the same type, except for division of `int`s. The divide operator applied to two `int`s returns an `option<int>` which contains nothing when the second operand is `0` and contains the truncated quotient otherwise.

5. `-`, `!` The unary negate and logical NOT operators. Negate takes in an `int` or `float` and returns the same type. Logical NOT takes in a `boolean` and returns a `boolean`.

6. `.`, Apply Dereferencing a field from an expression and applying arguments to a function. A dereference expression has the type of the member being dereferenced from the type of the left-hand side expression. Function application target types its arguments to the parameter types of the left-hand side function, and has as its own type the return type of that function.

Contained Expressions

Contained expressions include grouping parentheses, variable references, and seven types of literal. Their syntax is shown below:

```
contained
    : '(' expression ')'
    | NAME
    | INT
    | FLOAT
    | STRING
    | 'true'
    | 'false'
    | seqLiteral
    | mapLiteral
    ;

seqLiteral: NAME? '[' (expression ',')* (expression ','?)? ']';
```

[4]If the expressions have different types a warning will be issued.

```
mapLiteral: NAME? '{' (mapLiteralPair ',')* (mapLiteralPair ','?)? '}';
mapLiteralPair: expression ':' expression;

NAME: (UNICODE_LETTER | '_') (UNICODE_LETTER | [0-9_])*;

INT: [+-]? [0-9]+;

FLOAT
    : [+-]? [0-9]+  '.' [0-9]*                       // 1. 1.2
    | [+-]? [0-9]+ ('.' [0-9]*)? [eE] [+-]? [0-9]+   // 1e2 1.3e-4
    | [+-]?         '.' [0-9]+  ([eE] [+-]? [0-9]+)? // .1 .03e+1
    ;



STRING
    : '\'' (~['\\] | '\\\'' | '\\\\')*? '\''
    | '"' (~["\\] | '\\"' | '\\\\')*? '"'
    ;
```

Int, float, and string literals have the natural types `int`, `float`, and `string` respectively. `true` and `false` have the natural type `boolean`.

Sequence/Map Literals

```
[]
[1, 2, 3]
Array[]
Array[1, 2, 3]

{}
{'a': 1, 'b': 2}
Map{}
Map{1: 'a', 2: 'b'}
```

Sequence literals may be target typed to any type with the `[IsSequenceLike]` attribute and exactly one type argument, and map literals may be target typed to any type with the `[IsMapLike]` attribute and exactly two type arguments (for the keys and values in that order). Sequence and map literals only have a natural type if they have both a type name identifier in front of them and at least one element/key-value pair which has a natural type. Additionally, the type referenced by the type identifier must have the appropriate number of generic type arguments and must have the `[IsSequenceLike]`/`[IsMapLike]` attribute.

Primitive Types

MiKe has five primitive types, which all have the `[IsPrimitive]` attribute. These are:

- `unit` — A type with only one value. That value is unspeakable, though it may be obtained by storing the return value of a function which returns `unit`.
- `boolean` — A type with two values, `true` and `false`.
- `int` — Represents arbitrary-precision integers.
- `float` — Represents a double-precision IEEE 754 floating-point number.
- `string` — Represents a unicode string. The format is unspecified.

Standard Library

MiKe has a relatively small standard library which is designed to standardize basic functions while allowing the host to control the bulk of what is made available to developers. All functions and methods in the standard library could theoretically be implemented in at least amortized constant time,[5] though targets are not required to implement them as such.

Standard Library Types

The standard library types are shown below using an extended form of MiKe's syntax which shows attributes before the type and displays type parameters. Curly braces are used to indicate that the field/method set is not tied to a constructor.

```
[IsSequenceLike]
type Array<T> {
    get: (int) => option<T>,
    set: (int, T) => boolean,
    length: int
};

[IsSequenceLike]
type Queue<T> {
```

[5]Constructing objects from sequence/map literals may have to be implemented as linear in code side.

```
    enqueue: (T) => unit,
    pop: () => option<T>,
    peek: () => option<T>,
    length: int
};

[IsSequenceLike]
type Stack<T> {
    push: (T) => unit,
    pop: () => option<T>,
    peek: () => option<T>,
    length: int
};

[IsSequenceLike]
type Set<T> {
    add: (T) => unit,
    remove: (T) => boolean,
    has: (T) => boolean,
    length: int
};

[IsSequenceLike]
type QueueSet<T> {
    enqueue: (T) => unit,
    pop: () => option<T>,
    peek: () => option<T>,
    remove: (T) => boolean,
    has: (T) => boolean,
    length: int
};

[IsMapLike]
type Map<K, V> {
    set: (K, V) => unit,
    get: (K) => option<V>,
    remove: (K) => boolean,
    has: (K) => boolean,
    length: int
};

[IsLegalParameter]
[IsLegalCondition<destructInto = T>]
type option<T> {
    getOrDefault: (T) => T,
    hasValue: boolean
}
```

Standard Library Values

The MiKe standard library also provides a few values and functions to help write code. Two of these are `toInt: (float) => option<int>` and `toFloat: (int) => float` which help cast between numeric types. The other two are `some: <T>(T) => option<T>` and `none: option<?>` for constructing option values. These values are somewhat novel in that their types cannot be expressed using MiKe type syntax. `some` is a generic function whose return type depends on the argument type, and `none` is a generic *value* which is compatible with every other option type.

Appendix B: Expected Number of Events

When trying to calculate the expected number of events in a test input, it makes sense to simply focus on the expected number of events for a single user. After all, while one user's event may change the program state and thus alter the behavior of another user's event, the *validity* of each users' events are independent. Therefore, if we can calculate the expected number of events for a single user, we can just multiply it by six to get the expected number of events overall.

Let us begin with the case where there is no signal event handler, and we only have two join and two leave events. In this case, we can enumerate every possible combination of join (J) and leave (L) events.

J–J–L–L	L–L–J–J
J–L–J–L	L–J–L–J
J–L–L–J	L–J–J–L

Repairing these, we get the following event sequences (showing removed events with *):

J–*–L–*	*–*–J–*
J–L–J–L	*–J–L–J
J–L–*–J	*–J–*–L

From this, we can find the expected number of events when there are only join and leave events.

$$E(X_J) = \frac{1+1+2+2+2+1}{6} = \frac{9}{6} = 1.5$$
$$E(X_L) = \frac{1+0+2+1+1+1}{6} = \frac{6}{6} = 1$$
$$E(X_{JL}) = E(X_J) + E(X_L) = 2.5$$

We find that on average an activity will have $\frac{5}{8}$ of the maximum number of join/leave events ($\frac{2.5}{4}$), and that join events appear more commonly than leave events with a 2:1 ratio.

Once we introduce signal events, things become a little bit trickier. For each of the four signal events, there are five locations it can go: between two join/leave events, or before/after the join/leave events. Multiple signal events can also go to the same location, giving us 5^4

possible scenarios. However, we can avoid checking all of those scenarios by noticing that the expected number of events in each location is $\frac{4}{5}$ and whether or not a signal event in a particular location will be valid depends solely on the join/leave events around them. We can simply look at the six join/leave scenarios above and count the number of slots in which signal events are valid (S) versus invalid (/).

/–J–S–*–S–L–/–*–/	/–*–/–*–/–J–S–*–S
/–J–S–L–/–J–S–L–/	/–*–/–J–S–L–/–J–S
/–J–S–L–/–*–/–J–S	/–*–/–J–S–*–S–L–/

$$E(X_S) = \frac{2+2+2+2+2+2}{6} \cdot \frac{4}{5} = 2 \cdot \frac{4}{5} = 1.6$$
$$E(X_{JLS}) = E(X_J) + E(X_L) + E(X_S) = 1.5 + 1 + 1.6 = 4.1$$

So of the four possible signal events, we would only expect an average of 1.6 to make it into the final event sequence, and of the eight total events possible per user, on average only around half of them will actually be included in the sequence. Overall, we would expect the average event sequence to contain between 24 and 25 events.

Bibliography

[1] M. Grierson, "The codecircle platform," *The Routledge Research Companion to Electronic Music: Reaching out with Technology*, p. 312, 2018.

[2] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, "The drscheme project: an overview," *ACM Sigplan Notices*, vol. 33, no. 6, pp. 17–23, 1998.

[3] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, pp. 1–15, 2010.

[4] "Teams for education | replit docs," [Accessed 2023-04-04]. [Online]. Available: https://docs.replit.com/category/teams-for-education

[5] "Coding rooms - developer training and enablement," [Accessed 2023-04-04]. [Online]. Available: https://www.codingrooms.com/

[6] "Scratch 3.0 extensions," Jun. 2021, [Accessed 2023-04-06]. [Online]. Available: https://github.com/LLK/scratch-vm/blob/develop/docs/extensions.md

[7] M. L. Thomassen and M. Li, "Enterprise software implementation as context for digital innovation," *Selected Papers of the IRIS, Issue Nr 12*, 2021.

[8] R. B. Findler, "Drracket plugins," [Accessed 2023-04-06]. [Online]. Available: https://docs.racket-lang.org/tools/index.html

[9] T. Paley, "Necode: An environment for in-class programming activities," Worcester Polytechnic Institute, Tech. Rep., Mar. 2022.

[10] "p5.js web editor," 2023, [Accessed 2023-04-04]. [Online]. Available: https://editor.p5js.org/

[11] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs, second edition: An Introduction to Programming and Computing.* MIT Press, 2018.

[12] "Replit - kaboom draw," 2023, [Accessed 2023-04-04]. [Online]. Available: https://blog.replit.com/kaboomdraw

[13] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, 1989, pp. 399–407.

[14] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 2011, pp. 386–400.

[15] T. Jungnickel and T. Herb, "Simultaneous editing of json objects via operational transformation," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 812–815.

[16] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Yjs: A framework for near real-time p2p shared editing on arbitrary data types," in *Engineering the Web in the Big Data Era: 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings 15*. Springer, 2015, pp. 675–678.

[17] S. Weiss, P. Urso, and P. Molli, "Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks," in *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 2009, pp. 404–412.

[18] J. Fiala, M. Yee-King, and M. Grierson, "Collaborative coding interfaces on the web," in *Proceedings of the International Conference on Live Interfaces*. REFRAME Books, University of Sussex, 2016, pp. 49–57.

[19] B. Harvey and J. Mönig, "Snap! reference manual," 2020, [Accessed 2023-04-18]. [Online]. Available: https://snap.berkeley.edu/snap/help/SnapManual.pdf

[20] B. Garcia, F. Gortazar, L. Lopez-Fernandez, M. Gallego, and M. Paris, "Webrtc testing: Challenges and practical solutions," *IEEE Communications Standards Magazine*, vol. 1, no. 2, pp. 36–42, 2017.

[21] "1513107 - ondatachannel event will usually not fire for channel created after closing another channel," 2023, [Accessed 2023-03-23]. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1513107

[22] "Top ide index," 2023, [Accessed 2023-03-17]. [Online]. Available: https://pypl.github.io/IDE.html

[23] "Top ode index," 2023, [Accessed 2023-03-17]. [Online]. Available: https://pypl.github.io/ODE.html

[24] "Monaco editor," 2023, [Accessed 2023-03-23]. [Online]. Available: https://microsoft.github.io/monaco-editor/

[25] "Visual studio live share: Real-time code collaboration tool," 2023, [Accessed 2023-03-23]. [Online]. Available: https://visualstudio.microsoft.com/services/live-share/

[26] "Documentation - the zig programming language," 2023, [Accessed 2023-03-31]. [Online]. Available: https://ziglang.org/documentation/master/#if

[27] P. Lenkefi and G. Mezei, "Connections between language semantics and the query-based compiler architecture," in *Proceedings of the 17th International Conference on Software Technologies - Volume 1: ICSOFT.* INSTICC, 2022, pp. 167–174.

[28] "dubzzz/fast-check: Property based testing framework for javascript (like quickcheck) written in typescript," 2022, [Accessed 2023-02-27]. [Online]. Available: https://github.com/dubzzz/fast-check

[29] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000, pp. 268–279.

[30] M. Shaw, "Writing good software engineering research papers," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 726–736.

[31] "bestiejs/benchmark.js: A benchmarking library. as used on jsperf.com." 2023, [Accessed 2023-04-15]. [Online]. Available: https://github.com/bestiejs/benchmark.js/

[32] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[33] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren, "Port contention goes portable: Port contention side channels in web browsers," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 1182–1194.

[34] "Codemirror," 2023, [Accessed 2023-04-18]. [Online]. Available: https://codemirror.net/

[35] T. Parr, "The definitive antlr 4 reference," *The Definitive ANTLR 4 Reference*, pp. 1–326, 2013.