

A programming interface and toolchain for digital microfluidic biochips

Oliver Tobias Siggaard
Oliver Grønborg Christensen



Kongens Lyngby 2024
IMM-B.Sc.-2024

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-B.Sc.-2024

Abstract

This thesis presents the design, development and implementation of a visual programming interface for digital microfluidic (DMF) biochips. The primary objective is to create an intuitive toolchain that simplifies programming DMF biochips, particularly for users with limited coding experience. The visual programming language developed features a user-friendly graphical interface that allows for the creation of high-level protocols, using intuitively designed drag-and-drop blocks on a code canvas. These protocols are then to be converted into BioAssembly, executable by the DMF platform.

The toolchain ensures optimal droplet manipulation, avoiding unintended collisions and deadlocks, and supports operations such as moving, merging, splitting and mixing of droplets. The effectiveness of the solution is demonstrated through a suite of automated and manual tests, as well as a real-world case study involving a dilution series.

The results indicate a significant improvement in the usability of DMF platforms, especially for users without any prior experience in coding, facilitating more efficient and accurate biochemical experiments, lowering costs, and reducing reagent use.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer science of the Technical University of Denmark in fulfillment for the requirements of a BSC in Software Technology. The thesis was carried out from 29-01-2024 to 03-06-2024 under the supervision of Associate Professor Luca Pezarossa and Professor Jan Madsen.

Kongens Lyngby, 3-June-2024



Oliver Tobias Siggaard



Oliver Grønborg Christensen

Acknowledgments

First and foremost, we would like to express our gratitude to Luca Perzzarosa, our supervisor at the Department of Applied Mathematics and Computer Science, Technical University of Denmark. Luca's guidance, encouragement, and invaluable feedback has been crucial in helping us navigate the complexities of our project and achieve our goals.

We would also like to extend our thanks to our friends and family, for their support and patience throughout this journey. Your encouragement and understanding have meant a lot to us.

Thank you.

Contents

Abstract	i
Preface	iii
Acknowledgments	v
1 Introduction	1
1.1 Overview of Field	1
1.2 Motivation	2
1.3 Presentation	2
1.4 Outline	3
1.5 Source Access	4
2 Background	5
2.1 Microfluidic Biochips	5
2.1.1 DMF Platform	6
2.1.2 JSON Representation of the DMF Platform	7
2.1.3 BioAssembly	9
2.1.4 DMF Simulator	11
2.2 General Introduction to Compilers	12
2.3 Related Work	12
2.3.1 Visual Programming Languages	13
2.3.2 Graphical Interface for Digital Microfluidic Biochips	14
2.3.3 BioGo	16
2.3.4 Recap of Related Work	16
3 Technology Stack	19

4	Design	23
4.1	Objectives and Specification	23
4.2	Definition of Droplet Instructions	25
4.3	Overview of Data Flow	29
4.4	GUI	31
4.4.1	Initial Design Sketch	32
4.4.2	Final Layout and Functionality	32
4.4.3	Blocks	38
4.4.4	Block Editors	41
4.4.5	Input Validation and Restriction	42
4.5	Compilation and Execution Engine	46
4.5.1	Input and Output Format	47
4.5.2	Input Reader	50
4.5.3	Program Configuration	51
4.5.4	BioAssembly Generation	52
4.6	Handling Parallelization	54
4.6.1	Identifying Concurrent Actions	55
4.6.2	Pseudo-Concurrent Execution	56
4.7	Exception Handling	56
5	Implementation	57
5.1	Frontend	57
5.1.1	Block Data	58
5.1.2	Redux State Management	59
5.1.3	Code Canvas	62
5.1.4	Block Components	63
5.1.5	Block Editors	64
5.1.6	Sidebar	65
5.1.7	Program Sketch	65
5.1.8	Converting Blocks to Actions	67
5.1.9	Interaction with Backend	68
5.2	Backend	71
5.2.1	Point of Access - REST API	73
5.2.2	Input Reader	76
5.2.3	Droplets	80
5.2.4	Actions	83
5.2.5	Electrode Grid	88
5.2.6	A* - A Path-Finding Algorithm	92
5.2.7	Scheduling of Droplets	94
5.2.8	Compilation	100
5.2.9	Deadlocks	101
5.2.10	Exception Handling	102
5.2.11	Output Format	103

6	Testing and Evaluation	105
6.1	Automated Tests	105
6.1.1	Frontend	106
6.1.2	Backend	107
6.2	Manual Tests	110
6.2.1	Frontend	111
6.2.2	Backend	111
6.2.3	End-to-End Tests	111
6.3	Case Study: Dilution Series	112
6.4	Discussion	116
7	Conclusion	119
7.1	Summary	119
7.2	Future work	120
	Bibliography	123

CHAPTER 1

Introduction

This chapter provides a brief overview of the field of microfluidics, with a specific focus on digital microfluidic (DMF) biochips. The chapter discusses the motivation behind the project, and briefly presents the project in a short concise manner. Lastly, an outline of the thesis structure is provided, to guide the reader through the subsequent chapters.

1.1 Overview of Field

Microfluidics is a field of science and engineering, that deals with controlling and manipulating fluids at a very small scale, typically in the range of micro liters. These tiny volumes are handled within small micro-channels, using either external pressure sources or integrated mechanical micro-pumps [1]. It is a multidisciplinary field that involves engineering, biotechnology, microtechnology, and various other disciplines [2].

Microfluidic biochips, or lab-on-a-chip devices, allow for the automation, miniaturization and integration of complex biochemical protocols [1]. The integration of these devices can lead to significant reductions in sample and reagent volumes, increased assay speeds, lower operational costs and enhances scalability greatly. Digital microfluidics (DMF) is one of two subclasses of microfluidic systems,

where liquid is manipulated as small droplets on a two-dimensional array of electrodes. One of the most commonly used techniques, is Electrowetting-On-Dielectric (EWOD) [1], where voltages are applied on the electrodes in a predefined way, such that the desired droplet movement is achieved by creating/collapsing the electric field [2]. By activating adjacent electrodes systematically, operations such as moving, mixing, splitting and merging of droplets can be performed, as well as other operations traditionally performed in laboratories.

Microfluidic biochips have been used in a variety of applications, including drug discovery, diagnostic testing, prenatal screening and life science research [2], enhancing the efficiency and accuracy for scientists and researchers in the laboratory.

1.2 Motivation

Performing experiments on a tiny scale, as allowed by the digital microfluidic biochips, is a groundbreaking achievement. It allows for quick and controlled experiments, in a scale that could not have been achieved as easily before. However, it does not come without implications for the users, as the DMF biochips can be hard to program. The current language used to program the DMF biochips is BioAssembly [3], which can be hard to write and maintain, especially for biologists who may lack prior knowledge of computer programming. Therefore, a solution for quick visual programming of the DMF biochips in a graphical user interface is needed, motivating the first part of our project: The DMF-programmer, a web-based visual programming tool for DMF biochips. Additionally keeping droplets from colliding and ensuring that the optimal paths are taken, is not easy to visualize, regardless of the of the programming tool used. It is easy to lose track of the droplets when multiple droplets are at play at the same time. Therefore, a solution is needed to ensure proper and optimal paths for the droplets, motivating the second part of our solution: A compilation and execution engine that ensures proper and optimal movement of the droplets on the electrode grid.

1.3 Presentation

This thesis presents a programming interface and toolchain for digital microfluidic biochips. The programming interface should be a web-based graphical user interface (GUI) that is clear to understand and allows for easy programming of

the DMF biochips. The GUI should be user-friendly and provide code-blocks which can be added and dragged around on a canvas to give the user a visual programming experience. The toolchain consists of taking the program made by the code-blocks and converting them into BioAssembly. When converting the high-level program of blocks into BioAssembly, our toolchain should ensure optimal paths for the droplets, to avoid collisions between droplets and possible deadlocks. In the case of a stuck situation or an invalid program, the user should be informed with a descriptive error message in the GUI. Additionally, this thesis will present an evaluation of the final solution based on testing results, both from automated and manual tests, but also from a real-world case study in the form of a dilution series.

1.4 Outline

The structure of this thesis is organized into seven chapters, each providing detailed insights into various aspects of the research and development of the solution proposed in this thesis.

- **Chapter 2: Background**
Introduces microfluidics, focusing on digital microfluidic (DMF) biochips, related works, as well as concepts relevant for the context for the research in this thesis.
- **Chapter 3: Technology Stack**
Describes the technologies and tools used in the development of the visual programming interface and toolchain.
- **Chapter 4: Design**
Describes the design of visual programming interface and toolchain, including the user interface, design choices and data flow.
- **Chapter 5: Implementation**
Details the implementation process, covering front- and backend, and integration of components.
- **Chapter 6: Testing and Evaluation**
Outlines testing methods and results, including automated and manual tests, as well as presenting a case study to demonstrate the usability of the solution.
- **Chapter 7: Conclusion**
Summarizes the results and contributions of the thesis, as well as providing recommendations for future work.

1.5 Source Access

The source code for this project can be found in two separate GitHub repositories, one for the frontend and one for the backend:

- **Frontend** - bachelor-project-ui:
<https://github.com/OliverSiggaard/bachelor-project-ui>
- **Backend** - BachelorProject:
<https://github.com/OliverSiggaard/BachelorProject>

CHAPTER 2

Background

This chapter provides the background information which is necessary for the understanding of this project. We begin by exploring the DMF (Digital Microfluidics) platform, its operations and related work such as BioAssembly and the DMF Simulator. Following this, a brief introduction to compilers is made, drawing parallels to what we wish to achieve in this thesis. Our aim is to equip readers with a solid understanding of the microfluidics systems and compiler principles.

2.1 Microfluidic Biochips

Microfluidic biochips, also known as lab-on-a-chip devices, have significantly advanced the field of biotechnology, by enabling precise manipulation of small fluid volumes on small and compact platforms. The biochips offers various advantages, such as reduced reagent consumption, faster processing times, and allows to perform complex biochemical protocols with high precision. This section will provide a overview of the Digital Microfluidics (DMF) platform. We will start by examining the DMF platform developed at DTU, followed by an analysis of BioAssembly and the DMF Simulator, all of which will be utilized during this project.

2.1.1 DMF Platform

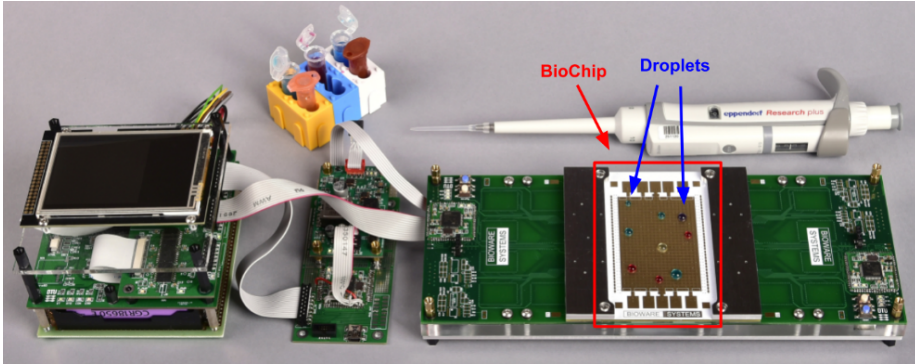


Figure 2.1: The DMF platform Developed at DTU [4].

The digital microfluidics platform developed by the Department of Applied Mathematics and Computer Science of Technical University of Denmark (DTU Compute) uses electrowetting on dielectric (EWOD) technology to control the movement of fluidic droplets on a chip. The chip, seen in Figure 2.1, consists of an array of individually addressable electrodes, that enables precise manipulation of droplets. This setup allows for easy reconfiguration of protocols, by simply altering the sequence in which the electrodes are activated or deactivated [4].

The modularity of the DMF platform, on both software and hardware level, allows for new platform modules to be developed and added to match any changes in the instrumentation needs of a biochip. This allows biochemists to rapidly reconfigure and iterate over multiple different droplet movement sequences, without physical alterations to the hardware. This flexibility can lead to significant increase in productivity, by reducing the time spent on setup and adjustments.

The DMF platform can be equipped with various sensors and heating elements, enabling it to function as a mini-computer, capable of monitoring a wide range of environmental and chemical variables. This integration, paired with a real-time execution engine, allows for real-time responses to changes in conditions, which can be crucial during sensitive biological reactions. Operating at a micro-scale, the platform also minimizes reagent use, making experiments more cost-effective and environmentally friendly.

2.1.2 JSON Representation of the DMF Platform

In addition to the physical setup and operation of the DMF platform, its configuration and state can also be represented using JSON (Javascript Object Notation). This structured, human-readable format is an agreement on how to represent the state of a DMF biochip in a file, making it interpretable and usable by computer programs. Such JSON representation format, hereinafter mentioned as DMFasJSON, has been developed at DTU Compute, along with its biochip.

The JSON representation is divided into several main JSON sections, seen in Listing 2.1. Each section represents different aspects of the platform.

Listing 2.1: Main sections present in DMFasJSON developed at DTU Compute

```

1 {
2   "information": [<json>, <json>, ...],
3   "electrodes": [<json>, <json>, ...],
4   "actuators": [<json>, <json>, ...],
5   "sensors": [<json>, <json>, ...],
6   "inputs": [<json>, <json>, ...],
7   "outputs": [<json>, <json>, ...],
8   "droplets": [<json>, <json>, ...],
9   "bubbles": [<json>, <json>, ...],
10  "unclassified": [<json>, <json>, ...]
11 }
```

For this thesis, the most relevant sections are the `information`, `electrodes` and `droplets` sections, which capture the information needed for controlling droplets on the microfluidic biochip.

Structure of select JSON sections:

1. Information

Listing 2.2: Detailed overview of the information section

```

1 {
2   "platform_name": <string>,
3   "platform_type": <string>,
4   "platform_ID": <int>,
5   "sizeX": <int>,
6   "sizeY": <int>
7 }
```

The JSON section seen in Listing 2.2 contains meta-information about the platform, most importantly the size of the platform in the X and Y axis respectively.

2. Electrodes

Listing 2.3: Detailed overview of an electrode representation in JSON

```
1 {  
2     "name": <string>,  
3     "ID": <int>,  
4     "electrodeID": <int>,  
5     "driverID": <int>,  
6     "shape": <int>,  
7     "positionX": <int>,  
8     "positionY": <int>,  
9     "sizeX": <int>,  
10    "sizeY": <int>,  
11    "status": <int>,  
12    ...other custom parameters...  
13 }
```

The JSON section seen in Listing 2.3 describes the individual electrodes present on the DMF platform, including their positions, sizes and current states.

3. Droplets

Listing 2.4: Detailed overview of a droplet representation in JSON

```
1 {  
2     "name": <string>,  
3     "ID": <int>,  
4     "substance_name": <string>  
5     "positionX": <int>,  
6     "positionY": <int>,  
7     "sizeX": <int>,  
8     "sizeY": <int>,  
9     "color": <string>,  
10    "temperature": <float>  
11    ...other custom parameters...  
12 }
```

The JSON section seen in Listing 2.4 describes the droplets currently present on the platform, including their positions, sizes and properties.

By utilizing JSON as a medium for describing information about the DMF platform, flexible software-tools such as DMF simulators and control systems can be created to fit different DMF platforms with varying configurations. In this project, DMFasJSON will be utilized, to read the dimensions of the platform, as well as information about the electrodes present on the DMF platform, to ensure compatibility with different platform configurations.

2.1.3 BioAssembly

BioAssembly, developed by DTU Compute, is an Instruction Set Architecture (ISA) specifically designed for digital microfluidics platforms. An ISA, in computer technology terms, is a set of instructions that a processor can execute, effectively defining how software can control the hardware. BioAssembly aims to standardize the command set across different digital microfluidic platforms, allowing for easier integration and more advanced programming languages and interfaces [3].

Important Instructions for this thesis

While BioAssembly consists of more than 40 different instructions, we will only be using a handful of these in this thesis. Table 2.2 lists the BioAssembly instructions that are particularly relevant for controlling the movement of droplets and the operations in our project.

Instruction	Description
SETELI	Set (activate) an electrode, attracting droplets to the electrode.
CLRELI	Clear (deactivate) an electrode, stopping it from attracting nearby droplets.
TICK	Apply changes and progress the system clock.
TSTOP	End the current task. Used for signalling the end of a program.

Figure 2.2: BioAssembly Instructions Relevant to Our Project

The instructions are especially important, for the precise control of moving droplets on the biochip. For instance, ‘SETELI’ activates a specific electrode, attracting droplets, while ‘CLRELI’ deactivates an electrode, stopping its influence on the droplets. The ‘TICK’ instruction applies the changes, and progresses the system clock, effectively controlling the flow of operations. Each BioAssembly instruction is separated by a semicolon.

Example of instruction usage in BioAssembly:

1. SETELI (Set Electrode):

- **Instruction:** ‘SETELI <ElectrodeID>;’
- **Example:** To set the electrode 5: ‘SETELI 5;’

2. CLRELI (Clear Electrode):

- **Instruction:** ‘CLRELI <ElectrodeID>;’
- **Example:** To clear the electrode 13: ‘CLRELI 13;’

3. TICK:

- **Instruction:** ‘TICK;’
- **Description:** Apply changes and progress the system clock.

4. TSTOP:

- **Instruction:** ‘TSTOP;’
- **Description:** Ends the current task (end of program)

Introduction to DTU DMF platform-specific commands

BioAssembly instructions are compiled into specific hardware commands, which can be sent directly to the target DMF platform. On the DMF platform developed at DTU Compute, these compiled commands are currently being sent through UART communication. UART (Universal Asynchronous Receiver-Transmitter) is a hardware communication protocol used for direct communication between devices. The commands are formatted as strings of text that start with an opcode, and terminate with ‘\r’ with a required spacing of at least 50 milliseconds between each command.

While the detailed hardware commands are beyond the scope of this thesis, it is important to understand, that BioAssembly abstracts away the complexities of hardware-specific programming, enabling standardized and efficient control of digital microfluidic biochips, if widely adopted.

2.1.4 DMF Simulator

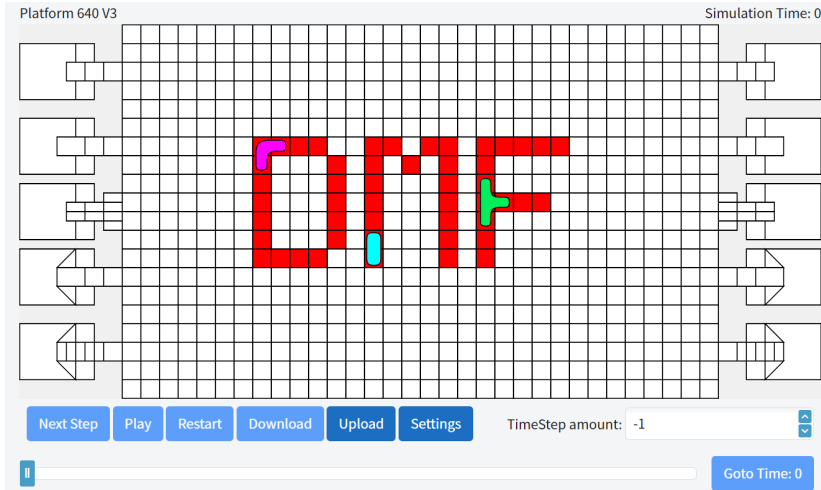


Figure 2.3: DMF Simulator user interface.

The DMF Simulator is an advanced web-application under development at DTU Compute by Alexander M. Collignon, Carl A. Jackson, and Joel A. V. Madsen, designed to simulate the behaviour of the actual DMF platform. Just like the real DMF platform described in Section 2.1.1, the simulator supports BioAssembly, allowing users to test their BioAssembly programs in a simulated environment, before executing them on the actual hardware.

Functionality and User Interaction

Users interact with the DMF Simulator through a Blazor-based web-interface, where they upload a JSON file containing the configuration of the target DMF platform, as well as a `.basm` file, containing BioAssembly instructions. The simulator then executes these instructions step-by-step, replicating the behaviour of the physical DMF platform. An overview of the user interface for the simulator, can be seen in Figure 2.3.

Benefits and Application in This Project

The DMF Simulator offers a significant advantage, by allowing programs to be executed, without the high costs associated with potential failures on the physical platform. This is particularly beneficial, when debugging and optimizing BioAssembly code. During the development phase of this project, the DMF Simulator was used extensively to visually verify the results of generated programs, ensuring that the generated programs align with the expected outcome.

2.2 General Introduction to Compilers

A Compiler is a specialized software tool designed to transform code written in one programming language (the source language) into another language (the target language). Traditionally, this process involves several stages such as preprocessing, lexical analysis, parsing, semantic analysis and code generation. The goal is to convert high-level instructions into machine-readable code, that can be executed by some target hardware.

In this thesis, we draw a parallel between traditional compilers and the system developed during this thesis. Instead of converting traditional text-based programming languages into machine code, we are transforming high-level droplet operations into BioAssembly, which can then be executed on the DMF Platform. This conversion allows for abstract, user-defined operations to be translated into concrete instructions which can be performed on the biochip.

In this thesis, we will present a Just-In-Time (JIT) compiler, compiling instructions on-the-fly in response to an emulated environment of droplets moving around on a biochip. This real-time compilation allows for dynamic decision making involving droplet path finding, avoiding unintended collisions and resolving of deadlocks. Additionally, our compiler performs optimization tasks similar to those found in traditional compilers. One key optimization is the scheduling of droplets, allowing multiple droplets to move simultaneously on the DMF platform. This concurrent movement of droplets improves the speed and throughput of protocols greatly.

Because of these parallels, we will refer to the process of converting higher-level droplet operations into BioAssembly as ‘compiling’, throughout the rest of this thesis.

2.3 Related Work

In this section we will review key contributions on the topic of creating a visual programming language for digital microfluidic biochips. We will discuss examples of general visual programming languages, the development of a visual programming language for microfluidic biochips, as well as how to process high-level languages into BioAssembly, which can be understood by the DMF platform. These works will lay the foundation for creating a comprehensive toolchain that integrates a visual programming interface, with a robust backend for compiling protocols into BioAssembly.

2.3.1 Visual Programming Languages

Visual programming languages (VPL's) allow users to create programs using graphical elements instead of traditional text-based code. This approach is beneficial for those with limited knowledge of traditional programming, and offers an intuitive and interactive way to develop software.

While there exists many examples of VPLs, we will briefly discuss two prominent projects in the following:

Blockly

Blockly is a web-based visual programming language developed by Google [5]. Programming using Blockly involves interlocking puzzle-like blocks, that represents various code structures, as seen in Figure 2.4. Users can easily rearrange these blocks by dragging-and-dropping them into place, to create functional programs. Blockly uses systematic colorings of blocks, to clearly distinguish between various block-categories such as control-flow or math blocks. The interface of the editor is minimalistic, with a block-selector in a sidebar to the left of the interface, a code-editor in the middle and a corresponding code-representation in the target language to the right. Blockly is primarily used as a educational tool for helping users learn programming concepts in a more engaging way, and is used by popular sites such as [Scratch.mit.edu](https://scratch.mit.edu) and [Code.org](https://code.org).

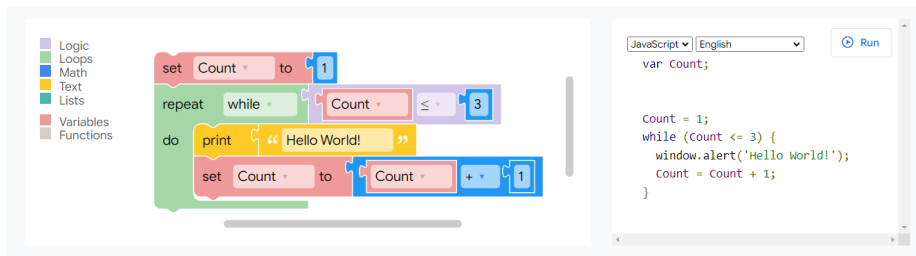


Figure 2.4: Example of a program in Blockly from their website.

LEGO Mindstorms

LEGO Mindstorms is LEGO's take on a re-programmable robot that can be easily programmed using a visual programming language. LEGO Mindstorms target a younger audience who want to learn programming and have fun with LEGO robots at the same time. There are several versions of LEGO Mindstorms, e.g., NXT and EV3, just to name a few.

Taking the software used to program the LEGO Mindstorms EV3 as an example. It features a ribbon-style programming interface, where users can drag and drop functions onto a ribbon/chain to create their program. The LEGO Mindstorms EV3 programming interface, along with an example program, is presented in Figure 2.5.

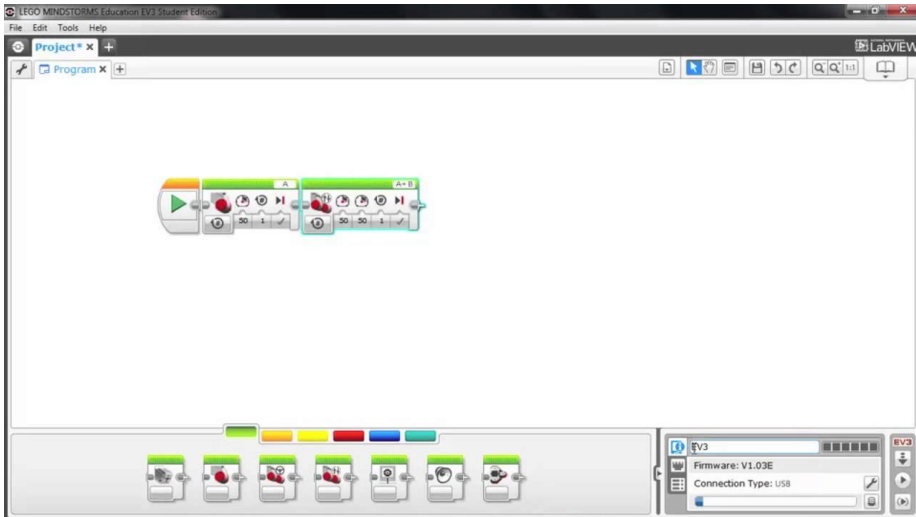


Figure 2.5: LEGO Mindstorms EV3 example program [6].

2.3.2 Graphical Interface for Digital Microfluidic Biochips

In the thesis ‘Design, development, and assessment of a graphical user interface for programming digital microfluidic biochips’ Jakub Ciok Wojciech presents a graphical user interface (GUI) for programming of digital microfluidic biochips.

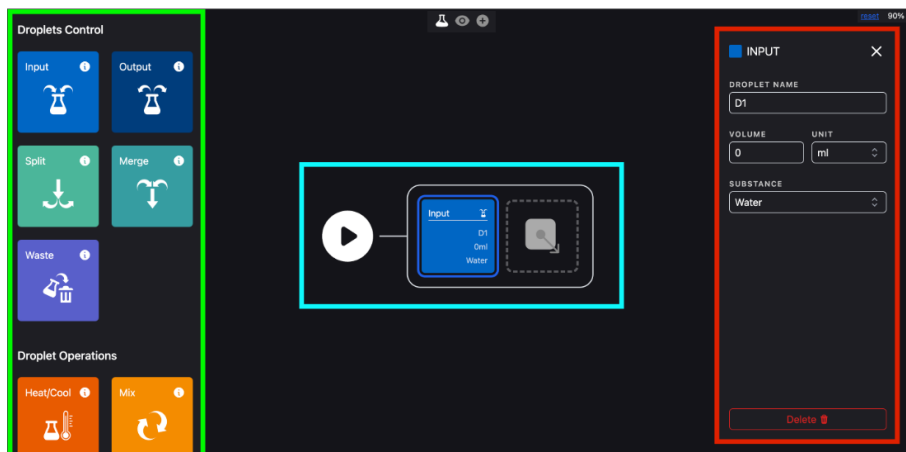


Figure 2.6: An overview of the GUI developed by Jakub Wojciech

The interface seen in Figure 2.6 includes a sidebar with high-level operations to the left (green), a canvas in the center (blue) and a settings panel to the right (red). The sidebar contains color-coded blocks representing various high-level actions on the platform. The color-scheme, as well as the icons representing the result of each block, helps the user to quickly identify and choose between relevant blocks. The canvas uses a ribbon-like style to visually indicate the order in which the actions are to be executed (left to right). The protocol can be modified, by dragging and dropping any existing blocks in the ribbon, or adding new blocks from the sidebar in the left of the screen. Each individual block in the protocol, can be selected, and relevant parameters can be modified using the settings-panel seen indicated by red in Figure 2.6.

The interface layout is intuitive, with clearly distinguishable blocks, that clearly indicate the results of each block. The most prominent parameters defined for each block can be seen directly on the blocks on the ribbon, omitting the need to individually inspect the configuration of each distinct block in the settings-panel.

One of the pit falls of this design, is that complex protocols tend to be very long, resulting in a very long sequence of blocks, where horizontal scrolling might be necessary, to see the entire protocol. The left-to-right flow of the blocks in the code canvas is intuitive for users accustomed to reading from left to right. However, this may be less intuitive for users from cultures where reading is traditionally done from right to left, such as certain Arabic-speaking regions, which necessitate additional visual cues, to indicate the order of execution.

2.3.3 BioGo

BioGo was initially presented in the thesis ‘Programming Biochemistry on Digital Microfluidic Biochips’ by Wenjie Fan. BioGo is a textual programming language that can be used to create protocols for digital microfluidic biochips, using a set of defined high-level operations. The language introduced by BioGo is consequently compiled into BioAssembly through a compilation process involving parsing programs from file to internal models, managing of droplet lifetime, and Just-In-Time compilation of a given protocol in an emulated environment. This process determines valid low-level operations in BioAssembly that correctly represents the provided protocol in the target language.

The conversion from BioGo into BioAssembly involves a Compiler and Executor, both implemented in C#, responsible for parsing from file to internal models and execution of operations from the aforementioned files respectively. The executor applies an instance of a A* algorithm to determine suitable paths for droplets, preventing unintended droplet collisions and effectively handling some deadlock situations.

A key feature of the BioGo executor, is that it allows for programs to be executed in synchronous and asynchronous mode respectively. While asynchronous mode simply relies on emulated data, synchronous mode introduces yet another layer of error-handling, by introducing a vision-based real-time detection system. This system uses camera technology to accurately detect the position of droplets on the biochip, allowing for adaptations during execution if the droplets do not behave as expected.

BioGo simplifies the process of writing protocols for digital microfluidic biochips, as it provides a high-level programming language as well as a robust compilation and execution framework. However, programming in BioGo may require some fundamental experience within the field of coding to be used effectively, which can pose difficulties for people with little to no coding experience.

2.3.4 Recap of Related Work

In summary, the discussed related works provide a good foundation for developing a user friendly interface for programming digital microfluidic biochips. The visual representation of protocols, as seen in LEGO Mindstorms and Jakub Wojciech’s GUI, demonstrates the effectiveness of using a ribbon-style programming interface. Additionally, it is important to note how the use of color-coded blocks and icons to easily distinguish between blocks help users easily identify and organize blocks into protocols.

The ability to convert these high-level visual instructions directly into BioAssembly makes it easier for biochemists to implement and execute protocols accurately. To achieve this, important aspects such as parsing of input, managing of droplet lifetimes, and preventing unintended droplet collisions through advanced path-finding algorithms must be used.

Building on these insights, this thesis aims to build upon the discoveries made by Jakub Wojciech and Wenje Fan, and develop a comprehensive toolchain for programming digital microfluidic biochips. By combining an intuitive visual programming interface with a reliable backend compilation and execution engine, we aim to create an easily accessible solution which can be used to simplify the programming and operation of digital microfluidic biochips, allowing for more efficient and accurate biochemical experiments.

CHAPTER 3

Technology Stack

In this chapter we will present the programming languages, frameworks, packages and other technologies that have been used in the development of this project. The software in this project has a clear distinction between front- and backend and that is no different when it comes to our technology stack.

Frontend

The frontend (GUI) of our project is a web-application built using TypeScript and React, along with other packages described here.

TypeScript and React

TypeScript [7] is a strongly typed programming language developed by Microsoft. TypeScript builds on JavaScript, but adds additional syntax for type annotations. As TypeScript is a superset of JavaScript it is also possible to write plain JavaScript in TypeScript. In fact, TypeScript converts code to JavaScript, which means that it can be run everywhere JavaScript can be run. The strong typing and the greater level of control, that results from this, is what motivated the decision of using TypeScript over plain JavaScript.

React [8] is a JavaScript/TypeScript library built by Meta (formerly Facebook). React is open-source and used for frontend web development. In React everything is a component, which allows the developer to break down the code into small reusable chunks providing solid ground for fast development and organized code.

npm and packages

Node Package Manager (npm) [9] is a software registry allowing developers to share and download packages. Once installed the user can use the Command Line Interface (CLI) to interact with npm and install packages to be used directly within the project. All of the packages mentioned below, which are used in the frontend were installed using the npm CLI.

- **Redux Toolkit** [10] - Redux is a JavaScript library for global state management, which can naturally also be used with TypeScript. Redux centralizes the applications state and the logic manipulating this state. Redux Toolkit (RTK) is a package for Redux used in this project which simplifies and enhances the standard Redux by providing a set of tools and best practices that aims to simplify using Redux as a developer [11]. We will be using Redux and Redux Toolkit synomonously in this thesis.
- **Axios** [12] - A popular JavaScript library used for making HTTP requests from web browsers. It simplifies the process of making API requests and handling the responses.
- **React-DnD** [13] - A package to help developers build drag and drop functionality into their React applications. This package eased the development of the drag and drop functionality of the code blocks on the code canvas.
- **Jest** [14] - A testing framework for JavaScript that also works with TypeScript. Jest allows for fast and easy writing of simple automated unit tests, with minimal configuration to get going.
- **Material UI** - MUI offers a wide variety of UI tools including pre-styled components with a wide variety of functionality, offering anything from drop-down menus to buttons and alerts [15]. MUI also offers an abundance of ready-to-use icons that have been used throughout our GUI [16].
- **Tailwind CSS** [17] - Tailwind CSS allows for direct styling inside the HTML and React components. It works by scanning for class names and converting these directly into CSS, this allows for faster development, as the developer can avoid explicitly writing the CSS.

Backend

The backend of our solution is a Java-based application, utilizing Maven as a project management tool, to simplify the installation of various dependencies described below.

Java

Java is a widely-used object-oriented programming language being used by millions of developers running more than 60 billion Java virtual Machines worldwide [18]. The backend is developed entirely in Java, giving us access to a wide array of well-documented and extensively used libraries described further hereunder.

Maven Dependencies

Maven serves as a project management tool, simplifying dependency management and build processes. This enables us to efficiently handle project configuration and dependencies, ensuring a smooth development process. Listed below are the key dependencies utilized in the backend for this project.

- **Spring Boot** [19] - A robust framework for building RESTful APIs in Java. Our usage of Spring Boot in this project, mainly revolves around creating a REST API that allows for communication between a Java-based backend, and a React/Typescript based frontend. REST stands for Representational State Transfer and is a set of guidelines to manage communication between applications, commonly used for web-services that communicate over the internet [20].
- **Jackson** [21] - Jackson is one of the most well-known and widely used open-source libraries for JSON processing in Java, being actively developed on GitHub. Jackson enables for easy conversion between JSON data and Java objects, allowing for seamless serialization and deserialization from JSON data to Java objects and vice versa.
- **Lombok** [22] - Lombok is a Java library that helps reduce boilerplate code, by automatically generating common methods such as getters, setters and constructors. This improves code readability by removing unnecessary clutter, and reducing the amount of repetitive code, that developers need to write.

- **JUnit** [23] - JUnit is a widely used testing framework for Java. JUnit provides annotations and assertion methods for writing unit tests, which is essential to ensure reliability and correctness in any application.

Git / GitHub

Git is a version control system that tracks changes to code, enabling collaboration and project management. GitHub [24] is a web-based hosting service owned by Microsoft, that allows for hosting Git repositories online.

During the development of this project, two separate GitHub repositories were utilized, housing our front- and backend applications. This setup emphasizes the modularity of our solution, while also allowing for a clear distinction between the front- and backend. Using Git and GitHub together in this project, has allowed for efficient collaboration, by enabling easy creation of multiple feature-branches, tracking of commit history, creation of pull requests, code reviews and much more.

CHAPTER 4

Design

This chapter details the design process of the visual programming interface for DMF biochips and toolchain developed throughout this thesis. It begins with a clear statement of the objectives and specifications that guided the design decisions. The chapter then describes the definition of droplet instructions necessary for manipulation of droplets on DMF biochips. An overview of the entire flow through the proposed system is provided, explaining how information from the graphical user interface is processed in the execution and compilation engine in the backend. We then dive into greater detail of the design of the GUI and the compilation and execution engine, addressing each separately. We round off the chapter by presenting the design choices regarding handling parallelized droplet movement and exceptions thrown during execution.

4.1 Objectives and Specification

The primary objective of this thesis, is to develop an intuitive visual programming interface and toolchain for digital microfluidic (DMF) biochips. The interface should simplify the process of programming DMF biochips, particularly for users with limited or no coding experience. The main objectives are as follows:

1. **User Friendly Interface:** Design and implement a graphical user interface (GUI) that allows users to create protocols with high-level operations, by dragging and dropping blocks in a code canvas. The interface should be simple and intuitive, allowing users to construct and manage protocols, without needing any prior experience writing traditional code.
2. **High-Level to Low-Level Compilation:** Implement a robust backend system, that is capable of translating series of high-level blocks from the GUI into low-level BioAssembly code, that can be executed on the DMF platform.
3. **Optimal Droplet Manipulation:** The toolchain should support essential droplet manipulation operations such as moving, merging, splitting and mixing. The system should optimize these operations, to avoid unintended collisions and deadlocks. Additionally, effective scheduling algorithms should be employed, allowing multiple droplets to move around on the platform concurrently if possible.
4. **Error Handling and User Feedback:** The system should be able to detect and efficiently handle any errors that occur during the protocol parsing and compilation processes, as well as directly in the GUI. The system should provide users with clear, descriptive error messages, that clearly indicate what actions should be taken, to resolve the issue.
5. **Evaluation and Validation:** The system should be thoroughly tested and evaluated through the use of automated tests, manual tests, and a case study of a real-world application of the solution. These tests should demonstrate the system's usability, accuracy and efficiency for making biochemical experiments.

By achieving these objectives, this thesis aims to improve the usability of DMF biochips, making them more accessible to a broader range of users, allowing for more efficient and accurate biochemical experiments to be performed.

4.2 Definition of Droplet Instructions

This section will introduce seven distinct droplet instructions supported by the visual programming language being developed in this thesis. Each instruction is essential for manipulating droplets on the digital microfluidic platform, enabling various different protocols to be created. The aforementioned operations can be seen on the following pages.

It is important to note, that all positions provided in the following examples refer to the top-left electrode of the rectangle bounding the droplet. More on the positioning of droplets can be found in Section 5.2.3.

Input Instruction

The **Input** instruction, as seen in Figure 4.1, allows the user to add a new droplet of fluid onto the platform. This is the initial step for any protocol, as it introduces the droplet(s) that will be manipulated in the subsequent operations. Without this instruction, there would be no droplets to process.

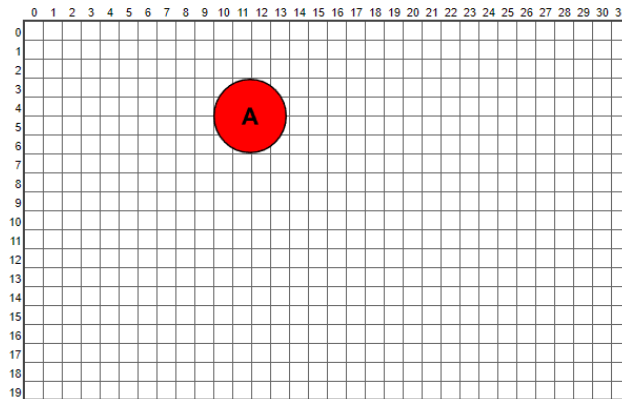


Figure 4.1: Droplet A is inserted at position (9, 3)

Output Instruction

The **Output** instruction, as seen in Figure 4.2, is used to move a droplet to a designated position, and mark it as unavailable for subsequent instructions for the rest of the execution. It is important to note, that the instruction does not physically remove the droplet from the platform; the droplet will still occupy its position on the platform.

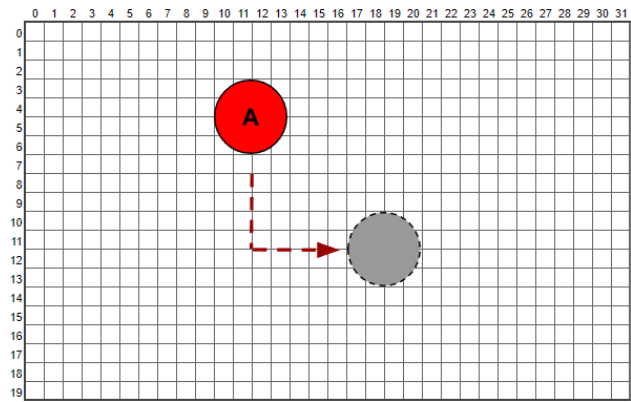


Figure 4.2: Droplet A is output at position (17, 10)

Move Instruction

The **Move** instruction, as seen in Figure 4.3, moves a droplet from its current position to a specified target position, ensuring that an optimal path is chosen and maintaining a safety distance of at least one electrode from other droplets. This is a essential operation that enables for the transportation of droplets on the platform.

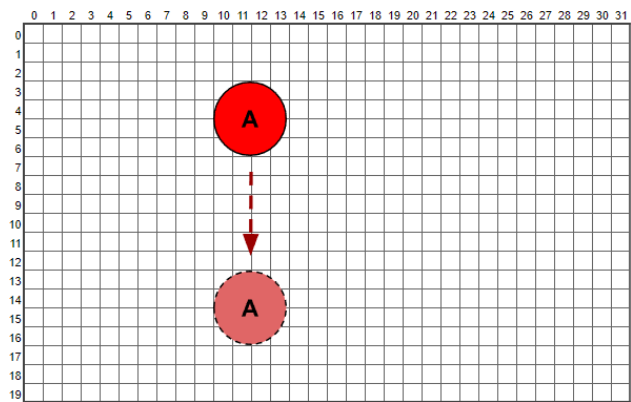


Figure 4.3: Droplet A is moved from position (3, 10) to position (10, 13)

Merge Instruction

The **Merge** instruction, as seen in Figure 4.4, allows two droplets to combine into a single droplet at a specified target position. This operation is essential in protocols that involve combining of reactants, for example, to initiate reactions or create new compounds for subsequent processing.

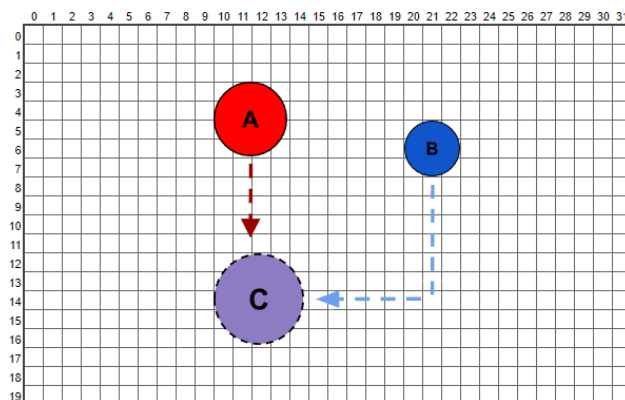


Figure 4.4: Droplet A and Droplet B is merged at position (10, 12)

Split Instruction

The **Split** instruction, as seen in Figure 4.5, divides a single droplet into two smaller, equally sized droplets, each moving to a specified target position. This operation is crucial in protocols that require distributing a fluid into smaller volumes, such as dilution series or preparing samples for parallel reactions.

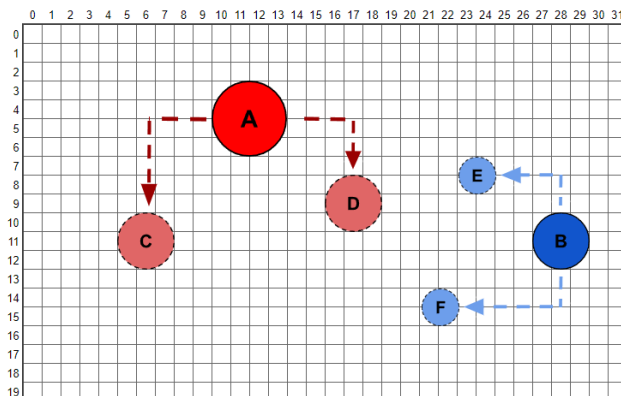


Figure 4.5: Droplet A splits horizontally into droplets C and D, moving to (5, 10) and (16, 8) respectively. Droplet B splits vertically into droplets E and F, moving to (23, 7) and (21, 14) respectively.

Mix Instruction

The Mix instruction, as as seen in Figure 4.6, allows for mixing of a droplet, by moving it in a rectangular pattern of a specified size, starting from a given position. This operation ensures thorough mixing of the droplets content, which is important for ensuring uniform solutions.

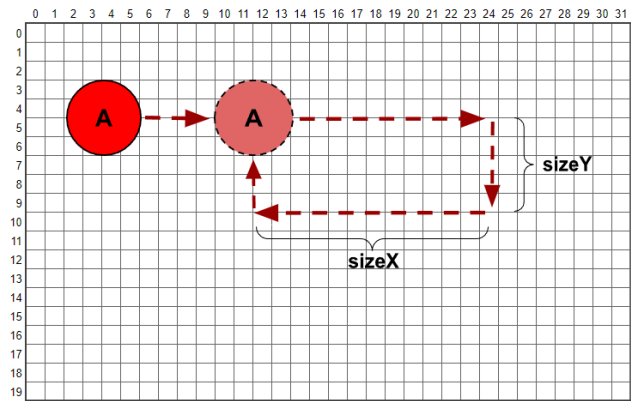


Figure 4.6: Droplet A is mixed from position (10, 3), following the shape of a rectangle with a width of 13, and a height of 5.

Store Instruction

The Store instruction, as seen in Figure 4.7, moves a droplet to a designated location and keeps it there for a certain amount of system ticks, making it unavailable for other instructions. This operation is useful for timed reactions or temporary storage of droplets, while other processes are completed.

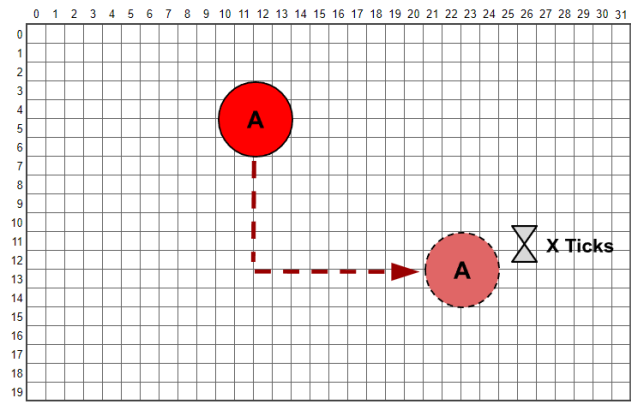


Figure 4.7: Droplet A is stored at position (21, 11) for x amount of system ticks.

4.3 Overview of Data Flow

A well-structured architecture is important for creating scalable, maintainable and efficient applications. By clearly separating the responsibilities between different components of the system, the development process is enhanced, leading to more organized and efficient workflows. In this project, we decided to utilize a split architecture, consisting of a frontend web-application and a backend Java application. This separation allows us to utilize the strenghts of each technology, while ensuring that the components can be developed and tested independently of each other. The frontend focuses on user interaction and the presentation layer, utilizing modern technologies and libraries for web development, such as React and Redux, to create an intuitive and responsive graphical user interface (GUI). The backend, on the other hand, handles core processing logic needed to compile protocols into BioAssembly, utilizing various libraries and frameworks available in Java's ecosystem, such as Spring Boot and Jackson. This modular approach ensures clear separation of concerns, and allows for a more efficient and organized development process.

This section provides an overview of the flow of data within the system, illustrating how the user can interact with the frontend and how information moves from the frontend to the backend and vice versa. The interactions between various components and how the user interacts with the system is depicted in Figure 4.8.

The process begins in the frontend, where the user is initially presented with a blank `Code Canvas`. Through the `Sidebar`, the user can then add blocks to the code canvas or change block information through the block editors and by dragging and dropping blocks on the `Code Canvas`, the user can rearrange the order of these. These direct user interactions manipulate the `Redux store`, which is the global state for our application. Block-based protocols can be downloaded or uploaded through the `Navbar` as so-called program sketches, allowing for easy sharing, continued development, and reuse of protocols at block-level.

When the user initiates the process of converting the block-based protocol to BioAssembly from the `Navbar`, the protocol is converted to a JSON-format which the backend understands and is then sent to the backend via an API request, where it will be processed by the `Input Reader` component. This component parses the JSON data and converts it into internal models that the `Executor` can then process. The `Executor` then attempts to run an internal simulation of the protocol based on these models. During execution, any errors that may be encountered, terminates the execution prematurely, and an appropriate error message is generated by the `Exeception Handler`.

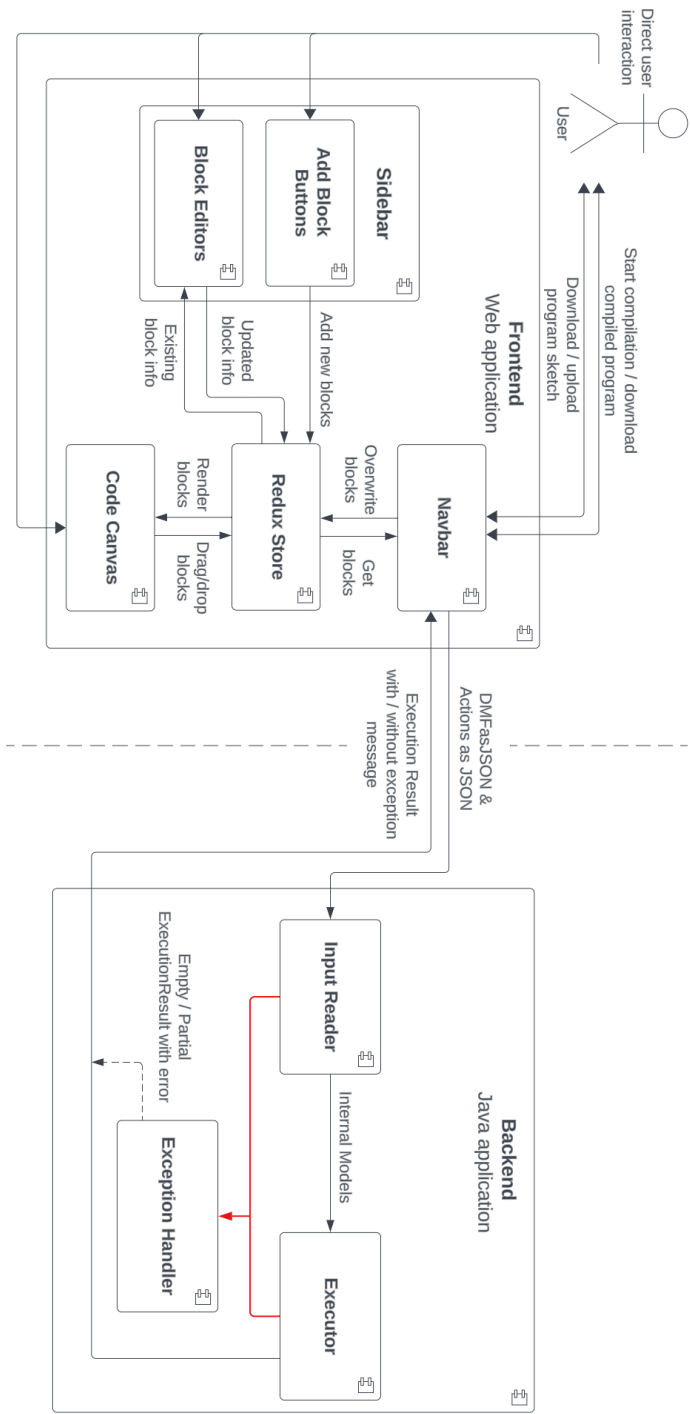


Figure 4.8: Flow of information at an abstracted level

The `ExecutionResult`, possibly including an error message, is sent back to the frontend. If no error occurred, an automatic download will begin of the BioAssembly-file for the compiled program and the JSON-file for the DMF platform configuration. If an error occurred, an error dialog with the relevant error message will be displayed. Depending on the error, a partial download of the un-finished BioAssembly protocol may be downloaded.

This toolchain allows users to visually create, edit and compile microfluidic protocols to be run on the DMF Platform, while effectively handling any errors that may arise during the execution process. The clear separation of responsibilities between the frontend and backend applications allows for a loosely coupled system, that not only can be developed independently of each other, but in fact also function independently of each other.

It is important to note that the diagram in Figure 4.8 abstracts the actual information flow. For instance, the `Navbar` is here depicted as having a lot of responsibility. While it is true that all the mentioned actions are triggered from, and somewhat handled by, the `Navbar` component, much of the functionality, such as converting the block-based protocol to the JSON format understood by the backend, is handled by utility functions. So while a lot of the flow goes through the `Navbar` because the buttons are placed there, we are in fact distributing much of the functionality to a series of helper functions, this is also true for most of the other components in the diagram.

4.4 GUI

This section will present the design choices made in regards to the graphical user interface. We will first present the initial design sketch of the GUI, followed shortly after by a presentation of the final version of the GUI, along with the layout and functionality of this. We will then take a deeper dive into the individual blocks that is used to program the protocol. In the end we will be discussing the choices, we have made when it comes to validating and restricting user input to reduce the amount of user errors.

4.4.1 Initial Design Sketch

At the very beginning of this project we created a first draft of the GUI in Figma [25], visualizing the ideas for our design. This sketch is presented in Figure 4.9.

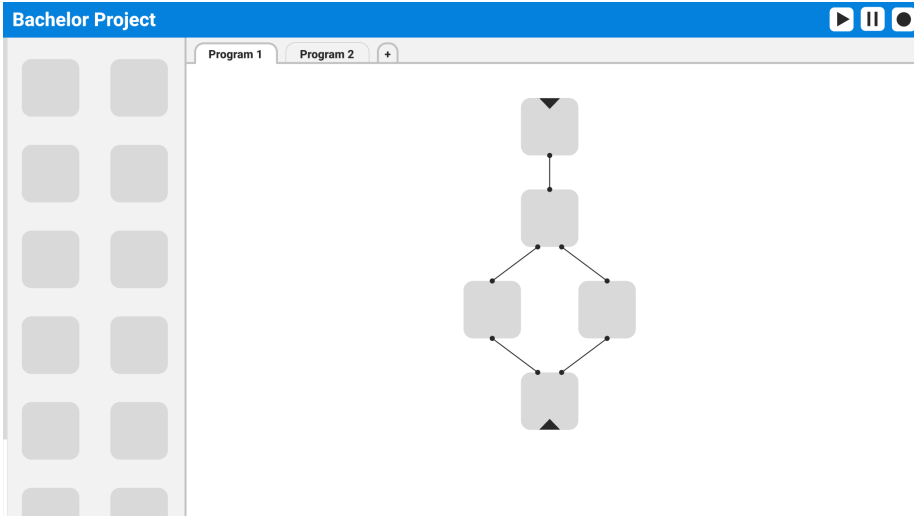


Figure 4.9: The first draft of the GUI

This first sketch gave us a great foundation and allowed us to agree on the direction in which our design should be headed. While providing great inspiration, the first sketch includes features that the final version does not have, while also lacking essential functionality. For example, the way in which the blocks are used to create the protocol is completely different. In fact, most aspects apart from the basic layout have been changed, underlining that this is just a sketch, that has allowed us to set the direction and start the design discussion.

4.4.2 Final Layout and Functionality

We will now present the final version of the GUI. A screenshot from our web-based programming interface for programming DMF biochips is presented in Figure 4.10.



Figure 4.10: The final version of the GUI

The layout of the GUI can be separated into three main components as highlighted by the color overlays in Figure 4.11. The three main components are the Navbar (blue), the Sidebar (red) and Code Canvas (green).

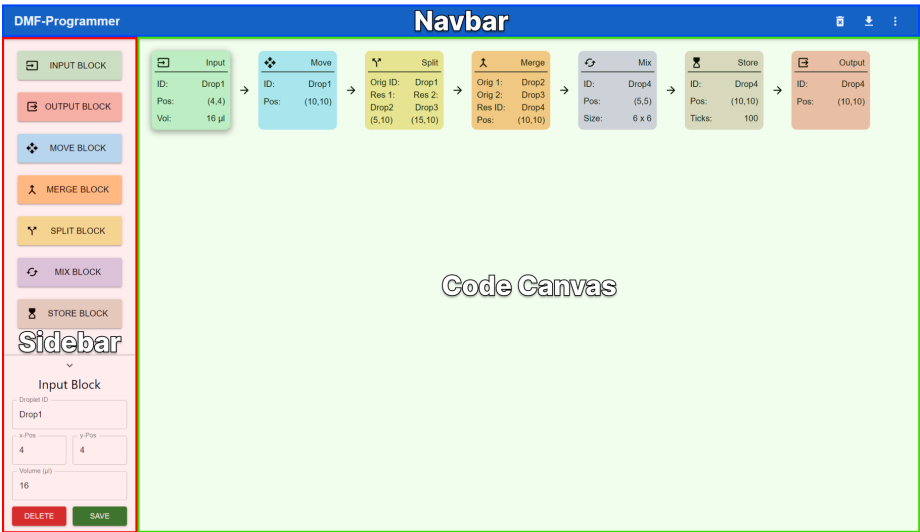


Figure 4.11: The main components of the GUI

We will now dive into each of the three main components, explaining the functionalities and other features embedded within.

Navbar

The navigation bar is placed at the top of the screen and is kept as simple as possible to ensure a minimalistic and modern design of the GUI while providing all the wanted functionality. The left-hand side of the navbar contains the web-applications name **DMF-Programmer**, while the right-hand side of the navbar contains three buttons as shown in Figure 4.12.



Figure 4.12: The navigation bar of the GUI

The first of the three buttons, which has an icon of a trashcan, allows the user to delete all blocks and start from scratch. When the user clicks on the button it will open a dialog, allowing the user to confirm or cancel their action and informing them of the consequences of deleting all blocks. This dialog can be seen in Figure 4.13.

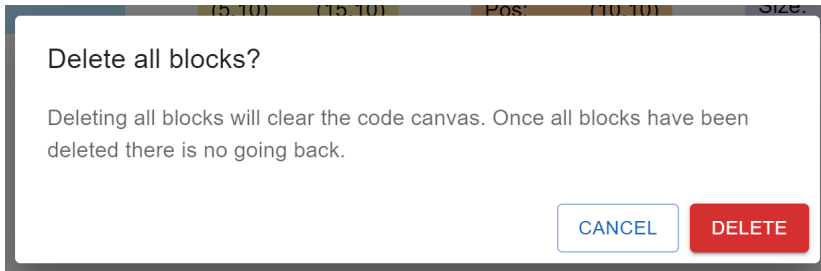


Figure 4.13: Delete All Dialog

The second button has a download icon and allows the user to download the program converted into BioAssembly, along with the platform configuration for the DMF platform. When the user clicks the button, a dialog will open, where the user can confirm or cancel the download. Furthermore, the dialog informs the user, that the program will be sent to the backend, where it will be converted into BioAssembly and that upon successful conversion, it will be automatically downloaded in the frontend along with the platform configuration. Because

usability is of high priority in the GUI, an additional message informs the users that they will need to allow automatic downloads in their browser. This is necessary because we will initiate the download of two files, and most browsers will ask for permission. The dialog is presented in Figure 4.14.

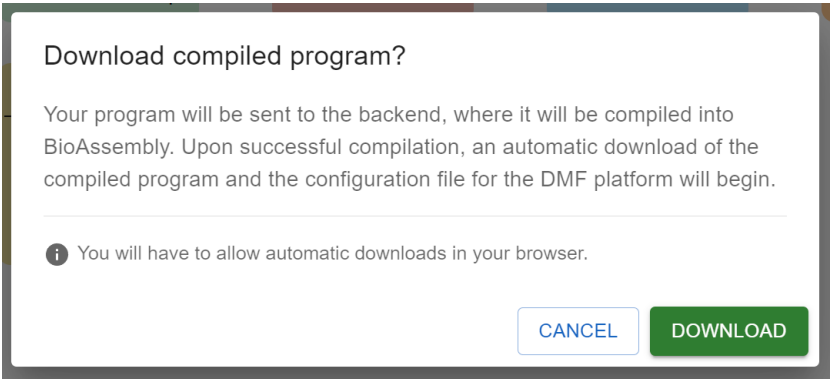


Figure 4.14: Download Dialog

The third button, which has an icon of three dots, is the ‘more’-button. When the button is clicked a drop-down list is presented as shown in Figure 4.15a. The menu has two options: downloading or uploading a program sketch. As mentioned earlier, a program sketch is a program that has not yet been converted into BioAssembly. This enables users to download and upload non-compiled protocols, including work-in-progress protocols in the block format. It enables users to easily share and switch between block-based protocols without discarding their current progress.

Because the buttons in the navbar are icons and not text, a tooltip will be shown when the user hovers a button, giving a brief textual description of that button. An example of this can be seen in Figure 4.15b.

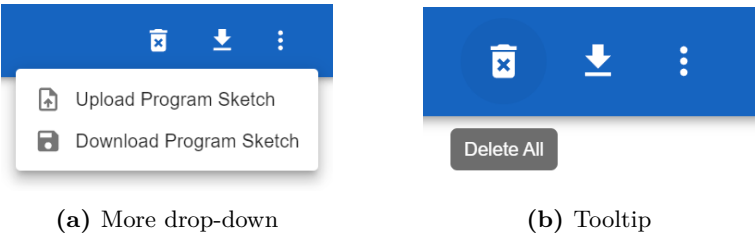
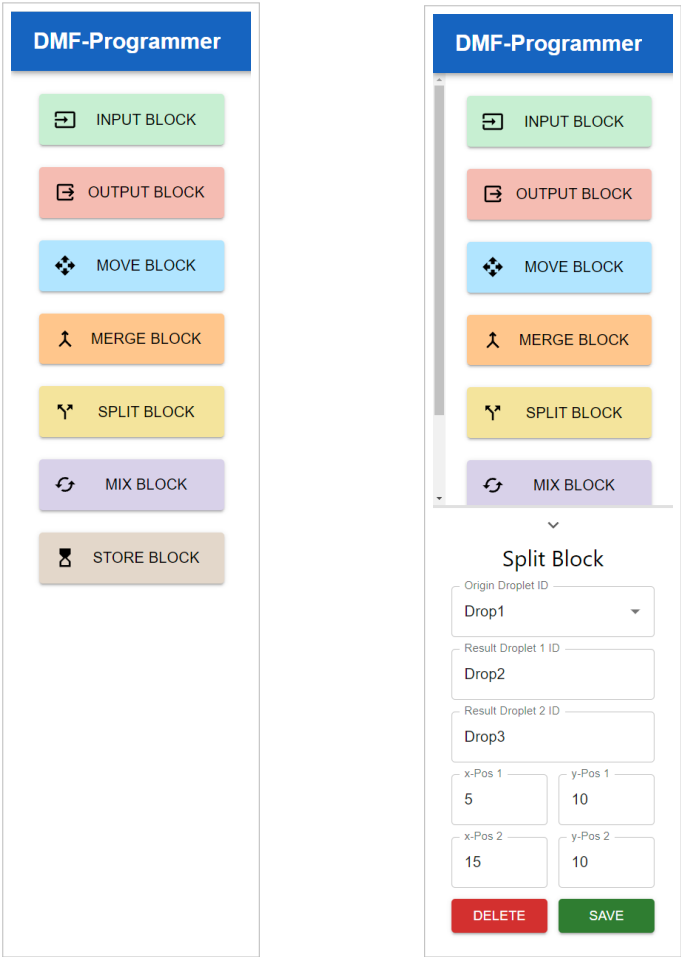


Figure 4.15: Details of the navigation bar

Sidebar

The sidebar is placed at the left-hand side of the web-page and has two main responsibilities: housing the buttons for adding code blocks to the code canvas and housing the editor for a code block, if one is selected. Figure 4.16a shows the sidebar without the block editor, as no block is selected. In contrast, Figure 4.16b shows the sidebar with the block editor for the selected block. In this case the editor for the split block is shown, which is the largest of the seven editors.



(a) Sidebar with no block editor (b) Sidebar with split block editor

Figure 4.16: The sidebar of the GUI, without and with a block editor

Depending on the user’s device, the editor might be overlapping the buttons to add blocks to the code canvas. We have designed the sidebar to cope with this overlap by enabling scrolling in the section containing the buttons, as indicated by the scrollbar to the left. This ensures that all buttons are accessible at all times to avoid compromising the user experience.

Code Canvas

The code canvas is the area that contains the code blocks. We will go into greater detail about the individual blocks in Section 4.4.3. For now, all we need to know is that code blocks are an essential part of the visual programming experience and that the order of these, along with the type and information stored within them, are what makes programming a protocol possible. When the user adds a code block, by clicking on a button in the sidebar, it spawns the code block at the end of the chain in the code canvas and opens the editor for that block in the sidebar.

A code block placed on the code canvas can be dragged and dropped to change its position relative to other blocks on the code canvas. A block can be selected by either clicking on it or dragging it on the code canvas. Selecting a code block will open the block editor for that specific block in the sidebar. The currently selected block is highlighted by the shadow behind it, visually lifting it off of the code canvas. Between each code block there are arrows indicating the flow of the program. If a chain of blocks becomes too wide to be on a single line, it will wrap around to the next, as shown in Figure 4.17.

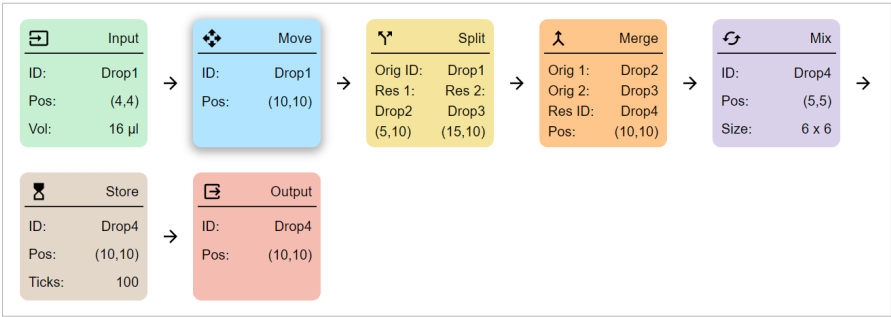


Figure 4.17: Code canvas close-up

4.4.3 Blocks

We will now present the seven distinct code blocks, each corresponding to the high-level droplet instructions introduced in Section 4.2. This time, we will delve deeper into the design of each block and the data stored in each of them.

Common for all code blocks is that they work on **droplet IDs**, a droplet ID is a unique ID given to a droplet that allows us to distinguish between the different droplets. In the case of code blocks, the droplet ID allows us to link a specific instruction to a specific droplet, by referring to the droplet ID from the code block.

Another common feature across all code blocks is that they contain at least one **position**. A position always has the form (x, y) , corresponding to an absolute position of an electrode in the electrode grid that constitutes the biochip. The position is zero-indexed and will always be the top-left corner of a droplet, which is represented as a square, more on this in section 5.2.3. It is not strictly necessary to include the position on every type of code block, as we have the move block to perform explicit move operations. However, we decided to include the position in all of the code blocks, to minimize the number of blocks a user needs to create in order to program a certain protocol. In practice, this means that we have built the functionality of the move block into the other code blocks. While this does make the move block less important, we still implemented the move block, as it will enable users to more precisely control droplet movement.

Input Block

The input block allows the user to add fluids to the platform. It is an essential block, as it is always the first block a user will have to place in order to have any fluids to work with. A close-up of the input block can be seen in Figure 4.18 and requires the following user input:

- A new unique droplet ID
- Input position
- Volume of droplet in μl

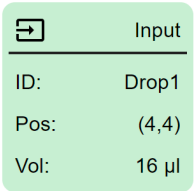
A light green rounded rectangle representing the input block. It contains a small icon of a box with an 'X' in the top left corner. To the right of the icon is the word 'Input'. Below this is a horizontal line. Under the line are three rows of text: 'ID: Drop1', 'Pos: (4,4)', and 'Vol: 16 µl'.

Figure 4.18: Input Block

Output Block

The output block allows the user to mark a droplet as removed from the platform. In practice this makes the droplet inaccessible for future code blocks, i.e. the droplet cannot be used anymore. A close-up of the output block is presented in Figure 4.19 and will require the following input from the user:

- Droplet ID
- Output position

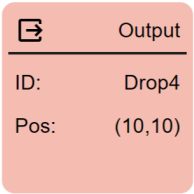


Figure 4.19: Output Block

Move Block

The move block allows the user to move a droplet without performing any other actions. The move block is presented in Figure 4.20 and requires the following user input:

- Droplet ID
- Target position

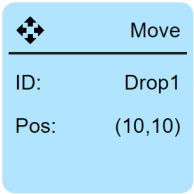


Figure 4.20: Move Block

Merge Block

The merge block allows for two droplets to be merged together to a single droplet. The merge consumes the two origin droplets and creates a new result droplet, which has a new and unique droplet ID. A close-up of the merge block is presented in Figure 4.21 and will require the following input from the user:

- Origin droplet ID 1
- Origin droplet ID 2
- Result droplet ID (new)
- Merge position


	Merge
Orig 1:	Drop2
Orig 2:	Drop3
Res ID:	Drop4
Pos:	(10,10)

Figure 4.21: Merge Block

Split Block

The split operation behaves contrary to the merge operation by consuming a single origin droplet and splitting it into two new result droplets, each with new and unique droplet IDs. Once the origin droplet has split, the two resulting droplets move to their specified target positions. The split block is presented in Figure 4.22 and requires the following input from the user:

- Origin droplet ID
- Result droplet ID 1 (new)
- Result droplet ID 2 (new)
- Target position of result droplet 1
- Target position of result droplet 2


	Split
Orig ID:	Drop1
Res 1:	Res 2:
Drop2	Drop3
(5,10)	(15,10)

Figure 4.22: Split Block

Mix Block

The mix block allows for mixing of a droplet by moving the droplet in a rectangular shape of a specified size. The mixing starts from a given position and ends in the same position after the mix has been completed. The mix block is presented in Figure 4.23 and requires the following user input:

- Droplet ID
- Start/end position
- Size (x × y)


	Mix
ID:	Drop4
Pos:	(5,5)
Size:	6 x 6

Figure 4.23: Mix Block

Store Block

The store block enables the user to move a droplet to a given position and mark it as inaccessible for a given amount of ticks. Essentially, this postpones all actions where the droplet would have otherwise been involved for the duration. The store block is presented in Figure 4.24, and requires the following input from the user:

- Droplet ID
- Store position
- Number of ticks


	Store
ID:	Drop4
Pos:	(10,10)
Ticks:	100

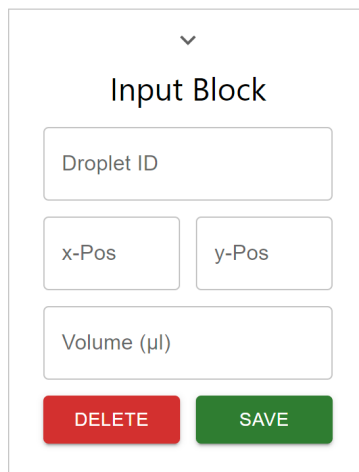
Figure 4.24: Store Block

4.4.4 Block Editors

Each of the seven blocks has their own block editor, which is placed in the sidebar as shown in Figure 4.16b. The block editor is what allows the user to edit the information of the blocks. All of the required user input mentioned previously in Section 4.4.3, is to be input through the block editors. Additionally, if the selected block already contains information, the block editor will be populated with that information when opened in the sidebar.

A close-up of the input block editor is presented in Figure 4.25. All block editors have three buttons: a red **Delete**-button which deletes the currently selected block, a green **Save**-button which saves the current changes made in the block editor and lastly the **Arrow**-button in the top of the editor, which closes the editor and deselects the currently selected block. To improve the user experience of our web-app, we have added some keyboard shortcuts to the block editors for faster and more convenient block editing. The three keyboard shortcuts correspond to clicking the three buttons mentioned above:

- **Delete block:** ctrl/cmd + backspace
- **Save changes:** ctrl/cmd + s
- **Deselect block:** escape



The image shows a user interface for editing an input block. At the top, there is a small downward-pointing chevron icon. Below it, the text "Input Block" is centered. The main area contains four input fields: "Droplet ID" (a single wide field), "x-Pos" and "y-Pos" (two side-by-side fields), and "Volume (μl)" (a single wide field). At the bottom, there are two buttons: a red "DELETE" button and a green "SAVE" button.

Figure 4.25: Input Block Editor

4.4.5 Input Validation and Restriction

The previous section described how users can update the information stored in the code blocks through the block editors in order to create a protocol. However, not all user input is valid, as some will lead to invalid protocols. Because usability is of high priority in the GUI, a series of input restrictions and validations, has been created in order to minimize the amount of user errors and catch them as early as possible. Not only do we want to restrict certain input, we also want to inform the users if they have to change something manually in order for the protocol to become valid.

The simplest restrictions that have been made are in regards to what characters the user is allowed to enter in the text fields of the block editors. We have restricted multiple text fields with regular expressions to only accept certain characters. This means that other characters simply cannot be entered, e.g. the minus symbol or letters cannot be entered if the field only allows positive integer values. We will go over which values some of the text fields have been restricted to in greater detail here:

- **Positions in all blocks** - All positions have been restricted to only allow positive integer values, including 0.
- **Input block volume** - The volume input in the input block editor has been restricted to only accept positive decimal numbers, including 0.

- **Mix rectangle size** - The rectangle size of the mix block editor has been restricted to only accept positive integer values, excluding 0.
- **Store ticks** - The amount of ticks in the store block has been restricted to only accept positive integer values, excluding 0.

The above are simple restrictions to avoid basic user errors, however, when it comes to droplet IDs and especially the availability of these, more logic is needed to ensure that users are informed about invalid droplet IDs. There are three reasons why a droplet ID might be invalid:

1. The droplet ID has never been created by either an input, split or merge block.
2. The droplet ID has been consumed by either an output, split or merge block.
3. The droplet ID is no longer available at a given position due to a change in the order of the code blocks on the code canvas.

If a droplet ID becomes unavailable as a result of one of the three reasons mentioned above, we inform the user by marking the droplet ID red directly in the code block on the code canvas. This gives the user a visual indication that a droplet ID is invalid and requires the user’s attention. An example of a move block with an invalid droplet ID can be seen in Figure 4.26, where the droplet ID ‘Drop1’ is red.

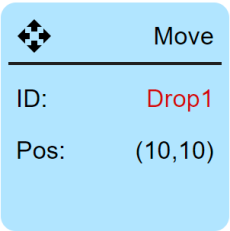


Figure 4.26: Invalid droplet ID

To avoid invalid droplet IDs from being entered in the first place, we have made it so that the droplet ID is selected from a drop-down menu. This drop-down menu contains only the droplet IDs that are currently available at the blocks position on the code canvas. Additionally, the drop-down menu features auto-complete, allowing users to search the list of droplet IDs, which is a particularly handy feature if many droplet IDs are available. An example of the auto-complete drop-down menu for selecting droplet IDs is presented in Figure 4.27. While this does in fact ensure that users cannot select invalid droplet IDs, there are still multiple reasons why a droplet ID can become unavailable, as described by the three reasons above. Common examples includes deleting an input block or moving an output block in front of the code blocks using the droplet ID that is being output. Both examples would result in unavailable droplet IDs which will then be marked as red.

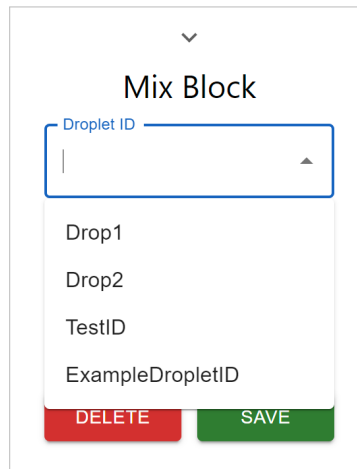


Figure 4.27: Auto-complete drop-down menu

Another problem that can arise in regards to droplet IDs is if user a tries to create a new droplet ID that already exists. In this situation, the user will not be allowed to save the block. Instead, when the user clicks the save button, a visual error will be displayed, marking the text field red and giving the user a textual description of the error. An example of this can be seen in Figure 4.28. Similarly, if the user tries to create two identical droplet IDs for the resulting droplets in a split block, the same error message is shown in both of the text fields.

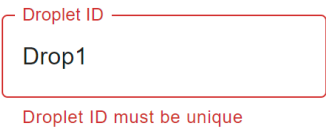


Figure 4.28: Non-unique droplet ID

However, it is not only when the user is creating new droplet IDs that we can encounter issues with non-unique droplet IDs. It can also happen if the user selects the same droplet ID to be *both* of the two origin droplet IDs in a merge block. In this case, a similar error is displayed to the user by marking the drop-down menu red and giving a textual explanation of the error, which informs the user that the origin droplet IDs are the same. An example of this error is presented in Figure 4.29.

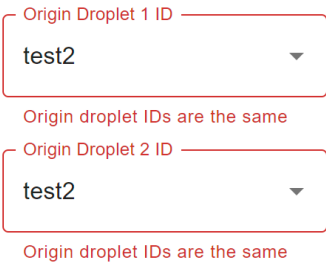


Figure 4.29: Same origin droplet IDs

While these input restrictions and validations are an important aspect of ensuring correct user input, they cannot alone guarantee that the protocol will execute without errors. Therefore, we also handle exceptions thrown in the backend, which we will go over in Section 4.7.

4.5 Compilation and Execution Engine

In this section, we will discuss the design and functionality of the compilation and execution engine. The purpose of this engine is to translate a list of high-level program-blocks, described in Section 4.4.3, into BioAssembly code, which can be executed on the DMF biochip. The backend performs Just-In-Time (JIT) compilation of these blocks, translating them into BioAssembly code that accurately depicts the intentions of the provided high-level protocol. During this process, the movement of each droplet is emulated internally in the backend, to ensure valid paths for each droplet, avoiding deadlocks and undesired droplet collisions.

We chose to implement this as a separate backend compilation engine instead of integrating it directly into the frontend for several reasons:

- **Separation of Concerns** By separating the logic revolving the conversion of high-level blocks into BioAssembly, we ensure a modular and more maintainable code-base. This loosely coupled approach allows each component to be developed and tested independently of each other.
- **Familiarity and Libraries** We chose Java for the backend, because it is well known to our team. This familiarity allowed us to develop and debug more efficiently. Additionally, Java offers a rich ecosystem of well-established libraries, some of which we will be utilizing in this project. More information about the libraries and frameworks used in this project, can be found in Section 3.
- **Strongly Typed and Object-Oriented Programming** While both Java and TypeScript are strongly typed languages that support Object-Oriented Programming (OOP), Java offered a few key advantages for our project:
 - **Performance** Java is a mature language, that has seen extensive performance optimizations throughout its releases. Additionally, Java has a proven record for housing large-scale, high-performance applications for many years, providing a well-understood and highly reliable environment for our backend.
 - **Robust Type system** Java's type system is stricter and more thorough than TypeScript's, providing better compile-time error checking, leading to reduced amount runtime errors. This is particularly important, when dealing with the complexities that the compilation and execution logic used in this project will entail.

4.5.1 Input and Output Format

The input and output formats are the only way for the backend to interface with the frontend, and vice versa. This modularity allows for easier development and testing of each component, by mocking the input and response from each counter-part. It is therefore important, to settle on a good interface (i.e. JSON structure), as all involved components will have to adhere to this structure in terms of types, parameter naming, and overall format.

The input to our compilation and execution engine is a JSON object, which can either be loaded directly from a file (mainly used for testing) or retrieved as a `String` through a REST API facilitated by Spring Boot. This JSON object contains a sequence of `Program Actions` that represents the operations to be performed on the DMF Platform. The `Program Actions` can be seen as a variation of what has formerly been described in the frontend as `Blocks`, more on this in Section 5.1.8. The structure of the JSON input is as follows:

Listing 4.1: The data format which the front- and backend agrees upon for sending protocols from the frontend to the backend. `dmf_configuration` section is explained in section 2.1.2

```

1 {
2   "dmf_configuration": {
3     "information": [<json>, <json>, ...],
4     "electrodes": [<json>, <json>, ...],
5     "actuators": [<json>, <json>, ...],
6     "sensors": [<json>, <json>, ...],
7     "inputs": [<json>, <json>, ...],
8     "outputs": [<json>, <json>, ...],
9     "droplets": [<json>, <json>, ...],
10    "bubbles": [<json>, <json>, ...],
11    "unclassified": [<json>, <json>, ...]
12  },
13  "program_actions": [
14    {
15      "action": "input",           // STRING
16      "id": "unique_action_id",   // STRING
17      "dropletId": "droplet_id", // STRING
18      "posX": 1,                 // INT
19      "posY": 2,                 // INT
20      "volume": 3,               // DECIMAL
21      "next": "action_id"       // STRING
22    },
23    {
24      "action": "output",         // STRING
25      "id": "unique_action_id",   // STRING
26      "dropletId": "droplet_id", // STRING
27      "posX": 1,                 // INT
28      "posY": 2,                 // INT
29      "next": "action_id"       // STRING

```

```

30     },
31     {
32         "action": "move",                // STRING
33         "id": "unique_action_id",        // STRING
34         "dropletId": "droplet_id",      // STRING
35         "posX": 1,                      // INT
36         "posY": 2,                      // INT
37         "next": "action_id"             // STRING
38     },
39     {
40         "action": "merge",                // STRING
41         "id": "unique_action_id",        // STRING
42         "resultDropletId": "result_droplet_id", // STRING
43         "dropletId1": "droplet_id1",    // STRING
44         "dropletId2": "droplet_id2",    // STRING
45         "posX": 1,                      // INT
46         "posY": 2,                      // INT
47         "next": "action_id"             // STRING
48     },
49     {
50         "action": "split",                // STRING
51         "id": "unique_action_id",        // STRING
52         "originDroplet": "droplet_id",  // STRING
53         "resultDroplet1": "result_droplet_id1", // STRING
54         "resultDroplet2": "result_droplet_id2", // STRING
55         "ratio": 3.5,                   // DECIMAL
56         "posX1": 1,                    // INT
57         "posY1": 2,                    // INT
58         "posX2": 3,                    // INT
59         "posY2": 4,                    // INT
60         "next": "action_id"             // STRING
61     },
62     {
63         "action": "mix",                  // STRING
64         "id": "unique_action_id",        // STRING
65         "dropletId": "droplet_id",      // STRING
66         "posX": 1,                      // INT
67         "posY": 2,                      // INT
68         "sizeX": 3,                    // INT
69         "sizeY": 4,                    // INT
70         "next": "action_id"             // STRING
71     },
72     {
73         "action": "store",                // STRING
74         "id": "unique_action_id",        // STRING
75         "dropletId": "droplet_id",      // STRING
76         "posX": 1,                      // INT
77         "posY": 2,                      // INT
78         "time": 3,                      // INT
79         "next": "action_id"             // STRING
80     }
81 ]
82 }

```

JSON objects of the form showcased in Listing 4.1 is to be generated in the frontend, and send to the backend for processing. In case the `dmf_configuration` section is not present in the received JSON object, a default configuration, corresponding to that of the platform seen in Figure 2.1 is used.

Each action in the sequence specifies the type of the operation (input, output, move, merge, split, mix or store), along with relevant parameters such as action ID, droplet ID's, positions, volumes and the ID of the action to be executed after the current action.

While most of the parameters such as droplet IDs, positions, volumes etc. are mapped one-to-one with the fields in the corresponding block showcased in Section 4.4.3, the following additional fields are included:

- **action** This is an identifier of the type of the action. For example, an action corresponding to a move block, has the value 'move'.
- **id** This is a unique identifier for each action, which is generated in the frontend, before being sent to the backend.
- **next** This is the ID of the next action to be executed after the current action is finished. This ensures proper ordering of the actions. This value is added in the frontend, before being sent to the backend.

The output of the compilation engine is an instance of the `ExecutionResult` class defined in the Java Backend. This class encapsulates the results of the compilation process, and contains the following parameters:

- **compiledProgram** A string, representing the compiled BioAssembly version of the program, generated from the provided sequence of program actions.
- **dmfConfiguration** A `JsonNode`-object that holds the DMF platform configuration, following the structure provided by `DMFasJSON`, explained in Section 2.1.2. More specifically, this version includes all droplets being generated by all `input`-actions, which allows the DMF Simulator (see Section 2.3) to visualize the droplets. However, this is not strictly necessary for the DMF Simulator, as it also capable to simulate just the electrode activity, without the droplets.
- **errorMessage** A string to store any error messages that may occur during the compilation process. This field is empty, unless an error is encountered during execution, in which case it is populated, and consequently displayed in the frontend.

Once generated, the aforementioned `ExecutionResult` instance is sent back to the frontend, where the object is deserialized, allowing each parameter to be read by the frontend.

4.5.2 Input Reader

Whenever a request is received from the frontend, the received JSON element (in the form described in Section 4.5.1) is processed by the input reader. While there is no class specifically named ‘`Input Reader`’ in the source code, as otherwise depicted in Figure 4.8, the term is used to describe everything involved in reading the raw JSON data, and converting it into usable models within the backend.

This process is designed to be highly expandable, allowing other data formats such as XML, YAML, etc., to be easily integrated into the system. Below is a brief overview of the process. For a more thorough explanation of how this is achieved or how to expand with another data format, see Section 5.2.2.

1. **Initial Raw-Data Interpretation** Upon receiving a JSON object from the frontend, the input reader maps the object to internal models representing both actions and platform information. This mapping process involves interpreting the raw data, and converting it into models that the backend can work with.
2. **Internal Models** The backend uses internal models to represent the parameters necessary for processing the data. These models include:
 - **Action Models:** Each action specified in the JSON objects seen in Listing 4.1 is mapped to a corresponding internal model that represents the higher-level operations to be performed.
 - **Platform Information Model** The DMF platform data (in the form of `DMFasJSON`) is mapped to an internal `PlatformInformation` class, where only select relevant data, such as the platform size and electrode information, is stored. This ensures that the backend has the sufficient data required, to understand the specific configuration of the DMF platform.
3. **Action Decomposition** Not all blocks are directly represented in the backend one-to-one. Some actions, such as the mix action, can be decomposed into a sequence of five (see Figure 4.6 for reference) `Move`-actions and are thus mapped to these simpler actions when converted to the internal models. This decomposition simplifies the implementation process, as multiple actions can be either partly or entirely decomposed into a sequence of simpler actions, making the execution of these more manageable.

The input reader's ability to interpret and convert raw data into structured internal models ensures that the backend can efficiently process and execute high-level program actions received from the frontend. This modular approach decouples the role of data interpretation and model initialization from the actual execution and compilation of the program. This separation allows each part to be developed and tested independently, improving maintainability and flexibility.

Lastly, the resulting `ActionModel` and `PlatformInformation` objects are consolidated into one object, called `ProgramConfiguration`. This object encapsulates all relevant data necessary for executing and compiling a given high-level program into `BioAssembly`. Additionally, this object is also responsible for maintaining relevant data structures such as an `ElectrodeGrid` (more on this in Section 5.2.5), which is initially derived from the aforementioned two datasources. This approach ensures that all necessary information is easily accessible and manageable during execution, all from a single object.

4.5.3 Program Configuration

The `ProgramConfiguration` class consolidates these `ActionModels` as well as the relevant data `PlatformInformation` into a single entity. In addition to this, the `ProgramConfiguration` also maintains key data structures such as the `ElectrodeGrid`, which represents the grid of electrodes on the DMF platform.

The `ElectrodeGrid` is derived from the size of the platform, as well as the list of `Electrodes` on the platform, which can be found in the `PlatformInformation` class. The grid corresponds to the actual layout of electrodes on the DMF platform, allowing for easy access to neighboring electrodes while moving droplets. If the electrodes are not placed in a rectangular grid, the compiler will deliberately throw an error, as dealing with arbitrarily shaped platforms is beyond the scope of this thesis.

To provide a clear understanding of how various operations are handled, it is important to mention the internal models of actions used in the backend. These are `InputAction`, `OutputAction`, `MoveAction`, `MergeAction`, `SplitAction`, `StoreAction` and `ActionQueue`.

The `ActionQueue` model decomposes complex actions into smaller, sequential actions to be executed one after another. The usecases includes the entirety of the high-level `Mix-block`, which can be decomposed into five `MoveAction`'s, and for the initial movement of droplets to the target position in `StoreAction` and `OutputAction`.

A detailed explanation of these internal actions and their implementation can be found in Section 5.2.4

4.5.4 BioAssembly Generation

With the internal models representing the high-level blocks, as well as the relevant platform information stored in suitable models in the backend, the next step is to generate the BioAssembly code, which corresponds to the protocol programmed using the high-level blocks in the GUI. This section will provide an overview of the process, which involves ensuring that a given program is executable, determining valid droplet movement sequences (if any), and finally converting the results into BioAssembly.

Schedule Generation

Initially, a **Schedule** is created from the list of **Actions** to be executed in the provided **ProgramConfiguration**. Internally, the schedule logic determines which actions should be performed at each tick based on the provided order of actions, the current status of each action, as well as the status of each droplet involved in said action. This ensures, that no actions are waiting unnecessarily long to begin execution.

In addition to this, the schedule also ensures consistency by verifying that no actions involving a specific droplet are executed before the droplet has been created and that each droplet in the program is created by exactly one action. If any inconsistencies are detected, the schedule will throw an error with an appropriate error message, halting the execution process. Further details on the implementation of the schedule, can be found in Section 5.2.7.

Execution Loop

Once a schedule has been established, the executor enters a loop, executing the actions retrieved from the schedule in each iteration. Each iteration corresponds to one tick in BioAssembly. The loop continues, until one of the following completion conditions are met:

1. **All Actions Completed:** If the schedule at any point in the execution is empty, this means, that all actions have successfully been executed.
2. **A Deadlock is Detected:** If at any point during the execution, a tick is reached, where no progress is made during the tick, we have entered a state of a deadlock. The program will now attempt to resolve the deadlock, as will be discussed in Section 5.2.9, or throw an error if no solution to the deadlock can be determined.

3. **An Error Occurs:** If any irregularities or non-resolvable states is entered during the execution, the loop will be terminated, passing down any relevant errors to be displayed in the frontend.

Path-finding and Collision Avoidance

During the execution loop, calculation of valid droplet moves is crucial to prevent unintended collisions between droplets. This process involves determining valid and optimal paths that droplets should take in a given tick.

Droplets must maintain a safe space of at least one electrode between itself and other droplets to avoid unintended merging. While move-actions require droplets to keep this safe distance, merge- and split-actions necessitate circumventing this safety distance. This is because droplets are allowed to collide when merging and in the initial ticks of a split action, while the droplets have not yet fully separated.

A detailed explanation of our solution for efficiently maintaining a safe distance between droplets can be found in Section 5.2.5.

Handling Termination Scenarios

Once the execution loop is complete, the system handles the result based on whether all actions were completed successfully or if it was prematurely terminated due to an error being thrown during execution.

In the case where all actions are successfully completed, the resulting ticks leading to this result are converted into BioAssembly. The primary operations that can occur in a tick are either no operations (e.g. a store action) or electrode activation/deactivation. These operations can be directly translated into its corresponding BioAssembly commands SETELI and CLRELI (Set Electrode and Clear Electrode) as described in Section 2.1.3.

In the case where the execution was prematurely terminated, a partial conversion is attempted, attempting to convert the results of all ticks leading to the point where the error occurred, into BioAssembly. This partially converted program, along with any relevant error messages, is sent to the frontend. This helps the user troubleshoot the root of the problem, by allowing the user to visually inspect the program's state leading to the error by running it, for example, on the DMF Simulator.

4.6 Handling Parallelization

In this section, we will discuss how the system handles the execution of pseudo-concurrent actions. The ability to execute multiple actions in the same tick is crucial for optimizing the performance and efficiency of the DMF platform.

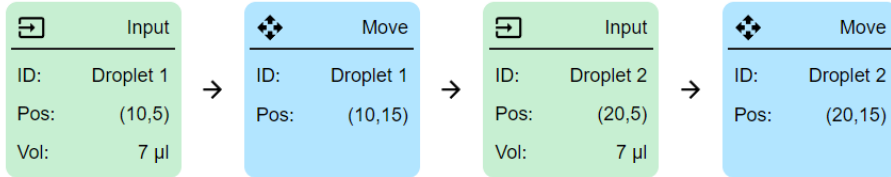


Figure 4.30: An example of a program, that would benefit from concurrent execution of actions.

Figure 4.30 illustrates the advantage of executing multiple actions within the same tick. Here, two droplets are inserted onto the electrode grid and then moved to different positions.

If we were to execute these actions sequentially, directly as provided by the block-based protocol in Figure 4.30, the sequence of actions would be:

1. Insert Droplet 1 at position (10, 5).
2. Move Droplet 1 to position (10, 15) - *spanning multiple ticks*.
3. Insert Droplet 2 at position (20, 5).
4. Move Droplet 2 to position (20, 15) - *spanning multiple ticks*.

This sequential execution would result in unnecessary waiting time, as each droplet could be moved concurrently without interfering with the other.

If the actions were instead executed concurrently, the sequence would be:

1. Insert Droplet 1 at position (10, 5) and Insert Droplet 2 at position (20, 5) in the same tick.
2. Move Droplet 1 to position (10, 15) and Move Droplet 2 to position (20, 15) concurrently - *spanning multiple ticks, both moving simultaneously*.

By executing the insert and move actions concurrently, the execution time in this example is cut in half. However, it is easy to imagine situations where the execution time can be reduced even further when more than two droplets are at play. This demonstrates the efficiency that can be gained through parallel execution.

4.6.1 Identifying Concurrent Actions

When polling actions from the `Schedule` that can be executed in the current tick, the schedule identifies all actions that can be executed concurrently. This involves ensuring the following criteria:

1. **Correct Ordering:** The order of actions requiring the same droplets must be maintained. For example, in Figure 4.30, the input actions must be executed prior to the move actions.
2. **Droplet Availability:** All droplets required to start the execution of an action must be available and not busy executing a previous action.

For any set of actions, which satisfies the aforementioned two criterias, the schedule should furthermore be able identify any actions that can (or cannot) be executed concurrently. Actions that can be executed concurrently, are exactly those, that do not require the same droplets for execution. For example, two droplets being inserted and subsequently moved to different locations, as it is the case in Figure 4.30, can be executed simultaneously, since there is no dependency between them.

Formally, if d_1 and d_2 denotes the set of droplets required for executing action 1 and action 2 respectively, the actions can execute concurrently if $d_1 \cap d_2 = \emptyset$. This constraint ensures that no two actions that satisfies this constraint, interfere with each other's droplets.

Additionally, as long as the ordering criteria are followed and no actions are attempted on droplets that haven't been created yet, it can, by induction, be induced that moving droplets satisfying all of the above mentioned criteria simultaneously, will not have any unforeseen impacts later in the execution.

The concurrent movement of droplet 1 and 2 in the ongoing example shown in Figure 4.30 will not cause any issues for the progress of either droplet further on in the program. According to the ordering criteria, the next actions affecting either droplet 1 or 2 are precisely those provided by the schedule, ensuring that no conflicts or dependencies are violated.

4.6.2 Pseudo-Concurrent Execution

During execution, multiple actions may be allowed to execute in a single tick, as determined by the `Schedule` described in the previous section. However, these actions are not executed concurrently in the traditional sense - at least not during the execution loop. Internally, each action is executed sequentially, prioritizing the first action in the list retrieved from the schedule¹.

Despite the sequential execution within the executor, the DMF platform perceives the results as if they were concurrent, since all commands being executed within a tick, are executed simultaneously on the platform. Thus, in the final results, droplets do appear to move concurrently on the DMF platform, even though their respective actions are executed sequentially during protocol compilation.

This approach has the significant advantage of avoiding real concurrency. By executing the actions sequentially, we eliminate the complexities associated with handling race conditions and synchronization issues, resulting in a simpler and more reliable system.

4.7 Exception Handling

As previously illustrated in Figure 4.8, exceptions thrown by the `Input Reader` and/or `Executor` are caught and processed by the `Exception Handler` to determine an appropriate error message.

When an error is encountered, the `Exception Handler` attaches an error message to either an empty or partial `ExecutionResult` object, and sends it back to the frontend. The frontend then checks if there is an error present in the execution result, or if the object only contains compiled `BioAssembly`. If an error message is present in the result, the user will be prompted with a dialog displaying the error, along with the option to download the partially compiled `BioAssembly` protocol, if one is available. This partial protocol, will include all actions that were successfully processed up until the point where the error was encountered.

As previously mentioned, the partially compiled program allows the user to simulate the program up to any potential point of interest where a problem was encountered, enabling the user to visually inspect the problem. This approach helps the user identify any issues, such as deadlock situations, and hopefully understand how to effectively modify the protocol to resolve the issue.

¹The mentioned prioritization is only changed in the case of deadlocks being detected. More about this in Section 5.2.9.

CHAPTER 5

Implementation

This chapter presents the implementation of the visual programming interface and toolchain for digital microfluidic biochips. It covers the implementation of both the front- and backend components, explaining how protocols are created, processed and compiled into BioAssembly code.

5.1 Frontend

In this section we will dive into the technical implementation of the graphical user interface. We will begin by giving an overall introduction to how the data of the code blocks are represented along with how the state of the code blocks are handled by Redux toolkit. We will then go over some of the individual components of the GUI and explain the implementation details of these. Following this we will explain how program sketches are downloaded and uploaded and what happens behind the scenes when a user either downloads or overwrites current progress. Lastly we will explain how the code blocks are converted to program actions which are the data format understood by the backend, along with the details of the API request made to the REST API in the backend.

5.1.1 Block Data

We will now explain the implementation details of the code blocks. At data-level a code block is a JSON object containing an index, a type and optionally some information of a type specific to the type of the code block. The indexes of the code blocks determines the sequence of the code blocks and is therefore important for the execution order of the protocol. The index is also important, as it is used when selecting a code block and thereby opening the block editor and being able to move the block around. The type indicates which of the seven types of code blocks the block is, which is important for multiple reasons, including rendering and validation. Lastly, the information is where the user input is stored in the code blocks at data-level. The info attribute is optional as the code block does not initially contain any information. To implement this structure of blocks we have utilized the strong typing of TypeScript and created an interface **Block** which is presented in Listing 5.1.

Listing 5.1: Block interface in frontend

```
1 export interface Block {  
2   index: number;  
3   type: string;  
4   info?: CodeBlockInfo;  
5 }
```

The info attribute of the **Block** interface has the type **CodeBlockInfo**. This is a union type, which means that it encompasses all of the block info types that we have defined for each of the seven block types. Not only does this allow for easy expansion if more code blocks were to be added. It also allows for each of the individual code blocks to fit under the single common type **Block** as defined by the interface. The **CodeBlockInfo** union type along with the seven interfaces for the different block types are shown in Listing 5.2.

Listing 5.2: Block Info Interfaces

```
1 export type CodeBlockInfo = InputBlockInfo | OutputBlockInfo |  
   MoveBlockInfo | MergeBlockInfo | SplitBlockInfo | MixBlockInfo |  
   StoreBlockInfo;  
2  
3 export interface InputBlockInfo {  
4   dropletId: string;  
5   posX: string;  
6   posY: string;  
7   volume: string;  
8 }  
9  
10 export interface OutputBlockInfo {  
11   dropletId: string;  
12   posX: string;  
13   posY: string;
```

```
14 }
15
16 export interface MoveBlockInfo {
17   dropletId: string;
18   posX: string;
19   posY: string;
20 }
21
22 export interface MergeBlockInfo {
23   originDropletId1: string;
24   originDropletId2: string;
25   resultDropletId: string;
26   posX: string;
27   posY: string;
28 }
29
30 export interface SplitBlockInfo {
31   originDropletId: string;
32   resultDropletId1: string;
33   resultDropletId2: string;
34   posX1: string;
35   posY1: string;
36   posX2: string;
37   posY2: string;
38 }
39
40 export interface MixBlockInfo {
41   dropletId: string;
42   posX: string;
43   posY: string;
44   xSize: string;
45   ySize: string;
46 }
47
48 export interface StoreBlockInfo {
49   dropletId: string;
50   posX: string;
51   posY: string;
52   time: string;
53 }
```

5.1.2 Redux State Management

Now that we have presented the format of the block data we are ready to present how they are stored in state. As mentioned in Section 3 where we present our technology stack, we are using a JavaScript library named Redux to manage a global state. Redux does, of course, also work with TypeScript, as it is a superset of JavaScript. Redux allows us to centralize the state of our application, as well as the logic for manipulating this state. All state regarding the code blocks is stored using Redux. This includes all block data of the form presented in Section 5.1.1, along with the index of the currently selected block.

An important feature to mention regarding our Redux setup is that we are using `persistStore` and `persistReducer`. This will make Redux use the local storage of the client's browser. Essentially, this means that the user can refresh, close and reopen the page without resetting the state. Because the state is saved locally and not on the server, it is client specific, which also means that the server hosting the web-application can go down or be reset without the state being lost for the client. This is a valuable feature as it helps users to not accidentally lose their work when they refresh or restart their browser or computer or if the web-server is taken down for maintenance or potentially crashes.

As described in the tech stack we are using a package called Redux Toolkit (RTK), which improves upon the developer experience of using the standard Redux. Most noticeably it reduces boilerplate code by combining state and logic in a so-called `slice`. Essentially, a slice in Redux Toolkit contains a name for the slice, the initial state and `reducers`. Reducers are important to understand because they are an essential part of Redux, as they are responsible for state management. A reducer is a function that specifies how the state changes in response to the actions dispatched to the Redux store. A reducer takes the current state and an action object and returns a new state manipulated by the action. The action can contain a payload used for manipulating the state, e.g. the index of the block to be selected. Redux Toolkit automatically creates action creators for each reducer allowing us to dispatch actions to the Redux store to change the state.

In our application we have one Redux Toolkit slice named `block`. The initial state is an empty array of blocks (of the type `Block`) and no currently selected block index. As presented in Listing 5.3, we have again utilized the strong typing of TypeScript and created an interface `BlocksState`, which defines the type of the state. We then create a constant `initialState`, which is of the type `BlocksState` and defines the actual initial state as described above.

Listing 5.3: Initial State

```
1 interface BlocksState {  
2   blocks: Block[];  
3   selectedIndex: number | null;  
4 }  
5  
6 const initialState: BlocksState = {  
7   blocks: [],  
8   selectedIndex: null,  
9 };
```


Reducers

Now that we have settled the basics of Redux, we present the many reducers in our code and their responsibilities in regards to manipulating the blocks state. Reducers are great because they allow for predictable and centralized state management and can be activated from anywhere in the code. A reducer is activated by using the `dispatch` function together with the action that we want to dispatch. This means that we can avoid prop drilling. Prop drilling is when data is sent through multiple components, which can become quite annoying, as it will often complicate the code unnecessarily. Dispatching an action from a React component directly to the Redux store, activating a reducer, allows us to minimize the amount of prop drilling in our application.

There is a reducer for all manipulations of the state concerning the code blocks, necessary to create a protocol in our block-based programming interface. The reducers are presented in list-format below:

- **addBlock** - adds a new block to the array of blocks in state.
- **removeBlock** - removes a block from the array of blocks in state.
- **moveBlock** - updates the position (index) of a block, along with all other blocks in the array, that needs their position updated as a result of the move.
- **deleteAll** - clears state and resets the state to the initial state.
- **editBlock** - updates a block's information as a result of user input from a block editor.
- **selectBlock** - marks the given index as the currently selected block.
- **overwriteBlocks** - overwrites state with a new array of blocks and de-selects the currently selected block, if any. This is used when a program sketch is uploaded.

Given the central role of reducers in our solution, we will explain in greater detail how the reducers are used throughout the application when we describe the parts of the application where they are utilized in the following sections.

Accessing state

While having a centralized state that is easy to manipulate is a great thing, it should also be easily accessible in the components that need it. Luckily Redux makes this easy by providing the `useSelector` hook. This hook takes a selector function as an argument and uses that to return only the portion of the state that is needed for the component. In our case we have two important parts of our state: the array of blocks and the currently selected index. These can be accessed through the `useSelector` hook as presented in Listing 5.4.

Listing 5.4: `useSelector` examples

```
1 const blocks = useSelector((state: {blocks: Block[]}) => state.blocks);  
2  
3 const blockIndex = useSelector((state: {selectedIndex: number | null})  
  => state.selectedIndex);
```

An important feature of the `useSelector` hook is that it makes the component automatically subscribe to updates to the state. This means that if the state is updated from a completely separate component, all subscribed components will re-render as a result of the update.

5.1.3 Code Canvas

The code canvas serves as the central component of the visual programming experience, as it contains the code blocks of the protocol that the user is using to program a protocol. By accessing the code blocks with the `useSelector` hook from our code canvas component we are automatically subscribed to any changes in the state, meaning that the code canvas will re-render every time there is a change in the array of code blocks. To render the blocks stored in the Redux state we are using the `map` function to iterate over the array of blocks, and for each block in the array we will call the `renderBlock`-function with the current block as the argument as shown in Listing 5.5.

Listing 5.5: Iterating over the array of blocks

```
1 {blocks.map((block) => renderBlock(block))}
```

The `renderBlock` function is a switch statement that, based on block type, calls a render function for that specific type of block. We have defined rendering functions for each of the seven block types, which has the responsibility of rendering the block of the given type along with the arrow to the right of the

block, unless it is the last block in the array. As an example we present the `renderInputBlock`-function in Listing 5.6.

Listing 5.6: Render function for input block

```
1 const renderInputBlock = (block: Block) => (  
2   <div className="flex flex-row items-center" key={block.index}>  
3     <InputBlock block={block} />  
4     {block.index !== blocks.length - 1 && <Arrow />}  
5   </div>  
6 );
```

5.1.4 Block Components

Each of the seven different types of code blocks have their own component, which is responsible for the type-specific layout and visual appearance of the code blocks. However, all seven code block components use a reusable wrapper component, `BaseBlock`. The `BaseBlock` component ensures consistent behavior and structure for all code blocks and has three main responsibilities which we will cover in the following.

First of all it ensures proper styling and structure of the code blocks, as it determines block size, margin, cursor style and the shadow if a block is selected, while leaving the type-specific styling to each of the individual block components.

Secondly, it handles block selection by dispatching the `selectBlock` action to the Redux store, whenever a block is clicked or dragged.

Lastly, the base block is responsible for handling the drag and drop functionality of the code blocks. This has been implemented using the React DnD library, which allows for easy implementation of otherwise quite complex drag and drop functionality. We have implemented it so that all blocks can be dragged while also serving as drop targets themselves. This is what allows the block being dragged to take the place of another block. From the users point of view, it has been made to look like the dragged block are pushing the other blocks directly on the code canvas. While this is true from a visual point of view, we are in fact updating the Redux state by refreshing all affected block indexes by the move. To be a bit more precise, we dispatch the `moveBlock` action to the Redux store with a payload consisting of the index of the dragged block and the index of the block over which the dragged block is hovering. This will update the state by removing the dragged block from its initial position, inserting it at the hovered position, and updating all other affected block indexes. It is important to note that this happens when the user hovers over a drop target, i.e. another

block. This means that the dragged block does not have to be released for the action to be dispatched. Because of the code canvas component's subscription to the Redux store through the `useSelector` hook, the correct placement of the blocks is always shown as the code canvas is re-rendered whenever the state changes. This allows for a responsive user experience where there is no doubt of the outcome of a block movement.

By using the `BaseBlock` component as a reusable wrapper, we are able to extract the common functionality of all code blocks to a single reusable component, reducing duplicate code and enforcing the DRY-principle.

5.1.5 Block Editors

The block editor is placed in the sidebar and is only rendered when a block is selected. We have extracted the logic for rendering the correct block to a separate component called `BlockEditor`. The `BlockEditor` component takes no parameters as it makes use of the `useSelector` hooks to access the Redux store and get the array of blocks along with the currently selected index. If a block is currently selected the `BlockEditor` will render the correct block editor based on the type of the currently selected block. E.g. if the currently selected block is an input block, the input block editor will be rendered. Because the `useSelector` hook makes the component subscribe to the Redux store, the correct block editor will automatically be rendered whenever a block is selected, without explicitly telling it to rerender.

The individual block editors, as presented in Section 4.4.4, have three buttons, namely the save, delete, and arrow button. The save button updates the state by dispatching the `editBlock` action to the Redux store with the user input as payload. The delete button updates the state by dispatching the `removeBlock` with the index of the block to be removed from the array of blocks. Lastly, the arrow button updates the state by dispatching the `selectBlock` action with the value of null, removing the selection of the current block. The deselection of a block will result in the currently open block editor to disappear, again as a result of the subscription to the Redux store.

5.1.6 Sidebar

Apart from housing the block editor, when a block is selected, the sidebar also contains the buttons to add new blocks to the code canvas, see Figure 4.16a. There is a button for each of the seven block types. When a button is pressed, the `addBlock` action is dispatched to the Redux store, with a payload consisting of the index and type of the block to be added. A block is always added to the end of the chain of blocks.

5.1.7 Program Sketch

We want users to be able to save their block-based protocols in a format that can be easily loaded into the Redux store, allowing them to continue their work at a later point in time or share it with a colleague. Luckily, this can be achieved quite easily, as the relevant data for this is simply just the array of blocks stored in state. In this section, we will explain how we download and upload the program sketch from and to the Redux store, respectively, as well as how we validate the uploaded program sketch to ensure that it can be rendered correctly.

Download

The program sketch is downloaded from the navigation bar, more specifically the *‘more’* drop-down menu. It works by simply retrieving the blocks from the Redux store with the `useSelector` hook and then converting the array of blocks to JSON. The downloaded program sketches are given time-specific names of the form `program_sketch_<timestamp>.json`, where the timestamp is of the format `hhmmss`, e.g. if the time is `11:45:23` the program sketch would be named `program_sketch_114523.json` allowing for easy distinction of multiple downloaded program sketches.

Listing 5.7: `downloadFile` utility function

```
1 export const downloadFile = (data: any, fileName: string, mimeType:
  string) => {
2   const blob = new Blob([data], { type: mimeType });
3   const url = window.URL.createObjectURL(blob);
4   const a = document.createElement('a');
5   document.body.appendChild(a);
6   a.href = url;
7   a.download = fileName; // Set filename
8   a.click();
9   window.URL.revokeObjectURL(url); // Clean up
10 }
```

To dive into the more technical aspects of the download we are using a utility function named `downloadFile`, which takes the data (content to be downloaded), the filename and the mime type of the content to be downloaded. For JSON the mime type is `'application/json'`. The `downloadFile` utility function is presented in Listing 5.7 and works by first creating what is called a `blob`, which is a chunk of data that can be manipulated as a file, e.g. downloaded. It then creates a URL representing the blob. To trigger the download, it creates an anchor element, sets the `href`-attribute to the URL of the blob, and sets the `download` attribute to the given filename. Lastly it simulates a click of the anchor element and revokes the URL. All of this simulates the automatic browser download of files. The same `downloadFile` utility function is utilized to download the compiled program and the platform configuration.

Upload

When we are uploading a program sketch, we are simply just overwriting the array of blocks stored in the Redux state and de-selecting any currently selected block. This is achieved by dispatching the `overwriteBlocks` action with a payload consisting of the new blocks.

The new blocks are created through a user-uploaded program sketch. The sketch is uploaded through an HTML input element of type `'file'`. The input element only accepts files of the mime type `'application/json'`. The default styling of the input element did not go well with our theme of the GUI, therefore we hid it from the user and instead created a label, which when clicked, triggers the file selection dialog of the input element. When a file is selected from the dialog it calls the `uploadProgramSketch` function as shown in Listing 5.8. In this function the file is retrieved and a `FileReader` object is created to read the file.

Upon finished reading of the file, the contents of the file is validated to ensure that the blocks will render correctly. If the contents of the file passes the validation the `overwriteBlocks` action is dispatched with the new array of blocks as payload. If the file does not pass the validation an alert is shown, giving the user a textual explanation of what caused the validation to fail.

Listing 5.8: uploadProgramSketch function

```

1 const uploadProgramSketch = (e: React.ChangeEvent<HTMLInputElement>)=>{
2   const file = e.target.files?.item(0);
3   if (file) {
4     const reader = new FileReader();
5     reader.onload = (event) => {
6       const fileContent = event.target?.result as string;
7       if (validateUploadedBlocks(fileContent) === true) {
8         dispatch(overwriteBlocks(JSON.parse(fileContent) as Block[]));
9       } else {
10        alert(validateUploadedBlocks(fileContent));
11      }
12    }
13    reader.readAsText(file);
14  }
15 }

```

We are validating indexes, types of the uploaded blocks, and if the block has any info, we are validating that the keys are correct. Essentially, we are checking whether each of the blocks will fit under one of our seven block types and that they will render correctly. This also means that we will not be validating the values of the information of the blocks, meaning that, even though it is less likely, users are still able to upload invalid programs, like they are also able to create invalid programs directly in the programming interface.

5.1.8 Converting Blocks to Actions

In the frontend we work with the concept of code blocks. As mentioned previously, these have a type, an index and optionally some information input by the user. However, in the backend, the concept of a code block does not exist in this format. Instead, there are **program actions**. Therefore, we will convert the blocks to the format of program actions as described in Listing 4.1. The attributes of the code blocks can more or less be directly mapped to those of the program actions as shown in Table 5.1.

Code Block	Program Actions
type	action
index	id
index + 1	next
info	info

Table 5.1: Mapping between code blocks and program actions

From the table it can be seen how the **type** of the code block, e.g. input, output or move, maps directly to the **action** attribute of the program action and how the index of the code block becomes the id of the program action. The code block does not have any concept of next, but since the blocks are placed in an array, it is simply just the next index. However, if the code block being mapped is the last in the array the next attribute of the corresponding program action will be set to '-1', which is not depicted in Table 5.1. Lastly, the **info** is the same between the two, the only difference being that the info of the code block is encapsulated in an **info**-object.

Comparing a code block and a program action representing the same data, it is clear to see the similarities and differences between the two. This comparison is presented in Listing 5.9.

Listing 5.9: Code block and program action comparison

```

1 // Code Block
2 {
3   "type": "input",
4   "index": 0,
5   "info": {
6     "dropletId": "test1",
7     "posX": "4",
8     "posY": "4",
9     "volume": "12"
10  }
11 }
12
13 // Program Action
14 {
15   "action": "input",
16   "id": 0,
17   "next": -1,
18   "dropletId": "test1",
19   "posX": "4",
20   "posY": "4",
21   "volume": "12"
22 }
```

5.1.9 Interaction with Backend

The block based protocol created by the user in the graphical programming interface can be sent to the backend to be compiled into BioAssembly. The interaction between the front- and the backend can be simplified to the following steps:

1. Retrieve code blocks from state.
2. Check that code blocks are present and that their info is filled out.
3. Convert code blocks to program actions.
4. Send request to the REST API in the backend, including the program actions as data.
5. Handle response from the backend, depending on the data contained in the received `ExecutionResult` object.

We have in previous sections covered how code blocks are retrieved from the Redux store using the `useSelector` hook, as well as how code blocks are converted to program actions. Therefore, we will in this section focus mostly on the implementation details of the API request.

Before sending the blocks to the backend, we do a basic validation of the blocks. First, we check if there are any blocks at all. Next, we check if all the blocks contain info and that all info fields are filled out. If any block fail these checks, the user is informed with an error message. If all blocks pass these checks, we convert them to program actions and send them to the backend via an API call. Note that this basic validation will not ensure that the protocol executes correctly, as that is a responsibility of the backend.

To handle the API call we have implemented a custom hook. This hook handles the state for `loading` and `success`, two boolean variables used to display in the GUI, whether an API call is in progress and if finished, whether it was successful or not, respectively. More importantly, the `useApiCall` hook contains a function `sendRequest`. This function allows the frontend to make API requests using `Axios`, which is a popular JavaScript library for making API requests. The `sendRequest` function takes four parameters:

- `url` - the endpoint of where to send the request.
- `method` - the HTTP method to use.
- `data` - the data to be sent in body of the request.
- `headers` - headers to be included in the request, e.g. the content type of the data to be sent.

The `useApiCall` hook provides the `sendRequest` component, along with the two boolean values `loading` and `success`, in the `Navbar` component where we are making the API call to the backend. The code snippet in Listing 5.10, shows how the `sendRequest` method provided by the `useApiCall` hook is used in the `Navbar`.

Listing 5.10: Usage of `sendRequest` method provided by the `useApiCall` hook

```
1 result = await sendRequest(  
2   "/api/compile",  
3   "POST",  
4   programActions,  
5   {  
6     "Content-Type": "application/json",  
7   }  
8 );
```

Here we can see that the url is `'/api/compile'`, which is the correct endpoint in the backend. The reason why the host and port is not included here is because this part is configured in the `Axios` configuration file to be the `baseUrl` of our Spring Boot backend application, which is set to be `'http://localhost:8080'`. The HTTP method is set to be `POST` as we are sending data to the server (backend). The data to be sent is stored in the constant `programActions`, which is the block-based program created by the user that have been converted to program actions. Lastly, the `Content-Type` header is telling the server that it is receiving data in a JSON-format.

The `sendRequest` function returns a promise in the form of a `ExecutionResult` object, which contains the compiled program, platform configuration, and any relevant error messages encountered during execution. If the backend server is not running, the API call will fail as expected, but this is caught and visually indicated to the user.

5.2 Backend

The backend of the project is the application responsible for processing high-level programming instructions and converting them into executable BioAssembly code, that can be executed on the DMF platform. This involves everything from parsing of data received from the frontend via the API, managing droplet movement, scheduling of actions, and resolving deadlocks. Figure 5.1 provides an overview of the key classes involved in this process, highlighting their relationships and roles in the process.

The main components illustrated in the Figure 5.1 include:

- **ProgramDataController:** This serves as the entry point for data being sent from the frontend, utilizing a REST API to handle incoming requests.
- **ProgramExecutionService:** This service processes the incoming data by parsing the received data into an instance of `ProgramConfiguration`, with the help of an implementation of the interface `IDtoToInternalMapper`, also annotated by `Input Reader` in Figure 5.1.
- **ProgramConfiguration:** Represents the configuration of the received protocol, including platform information, actions, and the droplets and electrode grid that can be derived hereof.
- **Executor:** The `Executor` is the core component that manages the execution and compilation process of the actions.
- **Schedule:** The `Schedule` manages the scheduling of droplets, ensuring that they are executed in the correct order without conflicts.
- **ExecutionResult:** When the compilation terminates, the results is encapsulated in a `ExecutionResult` instance, which includes the compiled program, DMF Configuration and DMF platform.

It is important to note that the diagram simplifies the overall architecture, and there are additional components involved in the backend implementation that are not depicted. These components along with more topic-specific diagrams, will be described in the sections below.

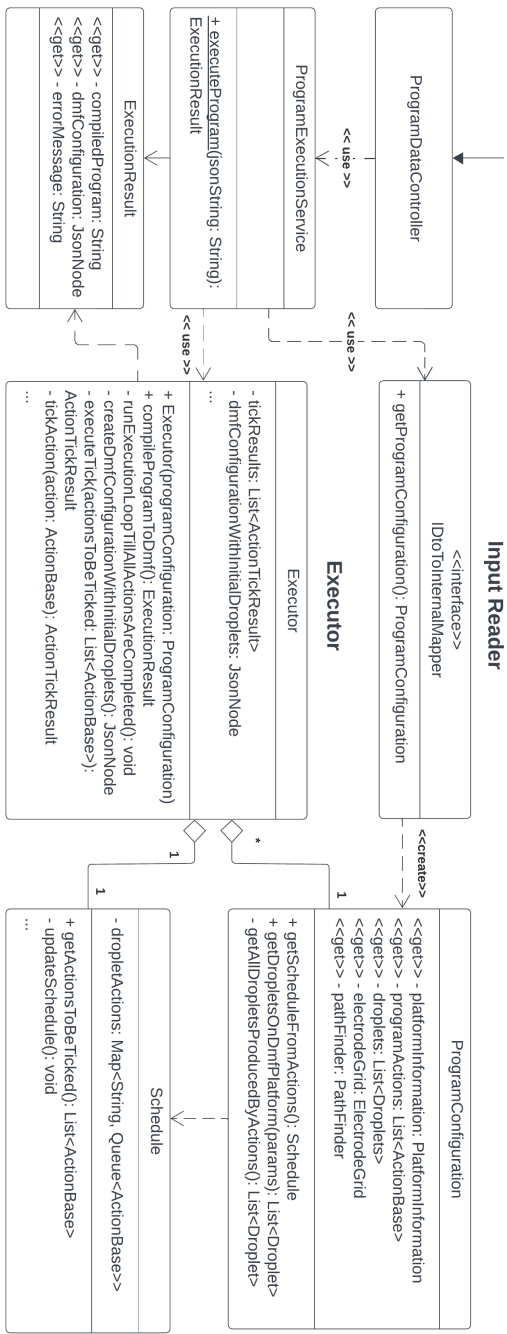


Figure 5.1: A very general class diagram, showcasing the most relevant classes. A lot of details have been omitted for simplicity. We have introduced the notion ‘<<get>>’ to indicate getters.

5.2.1 Point of Access - REST API

The primary point of access for the backend Java application, is a REST API implemented using Spring Boot. In this section, we will cover the details of the implementation of the REST controller `ProgramDataController`, which serves as the entry point for data received from the frontend.

Why Spring Boot?

Spring Boot was chosen for this project, because it is a widely used framework for creating RESTful web services. It greatly simplifies the setup of a REST API, which enables us to focus on the development of the business logic for the backend itself. The main reasons for choosing Spring Boot over other alternatives include:

1. **Ease of Configuration:** Spring Boot auto-configures the application based on which dependencies is added to the project, reducing manual setup and boilerplate code.
2. **Built-in Support for REST APIs:** Spring Boot offers useful annotations to define endpoints and manage request/response handling.
3. **Integration with other Technologies:** Spring Boot offers integration with many other Java libraries, such as Jackson for JSON processing, which is heavily utilized in this project, for serializing and deserializing JSON.

Implementation

The `ProgramDataController` class seen in Listing 5.11 uses key Spring annotations, to handle HTTP requests, manage cross-origin resource sharing (CORS), and define request URL mappings:

- **@RestController:** This annotation tells Spring, that this class will handle web requests, and automatically convert the return values of its methods to JSON format, simplifying the creation of the REST API.
- **@CrossOrigin:** This annotation allows web pages from one domain (in our case `'http://localhost:3000'`, which is where our frontend is running) to make requests to another domain (in our case, the backend server). It is necessary for enabling the frontend, which runs on a different origin, to communicate with the backend, without encountering security issues related to cross-origin resource sharing (CORS).

- **@RequestMapping:** This annotation specifies the base URL for all endpoints in this controller. In this case, the base URL is set to `‘/api’`, meaning all the endpoints in the controller, will start with `‘/api’`.

Listing 5.11: Code Snippet of ProgramDataController with only select parts of the code displayed

```
1 @RestController
2 @CrossOrigin(origins = "http://localhost:3000")
3 @RequestMapping("/api")
4 public class ProgramDataController {
5
6     @PostMapping("/compile")
7     public ResponseEntity<ExecutionResult> dataFromFrontend(
8         @RequestBody String data) {
9         HttpHeaders headers = new HttpHeaders();
10         headers.add(HttpHeaders.CONTENT_DISPOSITION, "attachment");
11         headers.setContentType(MediaType.APPLICATION_JSON);
12
13         try {
14             ExecutionResult executionResult =
15                 ProgramExecutionService.executeProgram(data);
16
17             if(executionResult.hasError()) {
18                 return ResponseEntity.badRequest()
19                     .headers(headers)
20                     .body(executionResult);
21             }
22
23             return ResponseEntity.ok()
24                 .headers(headers)
25                 .body(executionResult);
26         } catch (Exception e) {
27             /*
28              * Handle error, and return ExecutionResult object
29              * with an appropriate error message
30              */
31         }
32     }
33 }
```

ProgramExecutionService

The `ProgramExecutionService` class seen in Listing 5.12 is a service component in Spring Boot, annotated with `@Service`. A service in Spring, is a class that contains business logic, and can be used by controllers to perform specific tasks. Using services, helps in keeping the controller code clean, and separate business logic from the logic revolving around receiving and sending data.

Listing 5.12: ProgramExecutionService.Java

```
1 @Service
2 public class ProgramExecutionService {
3     public static ExecutionResult executeProgram(String jsonString) {
4         try {
5             // Convert json to programConfiguration
6             ProgramConfiguration programConfiguration =
7                 new JsonToInternalMapper(jsonString)
8                     .getProgramConfiguration();
9
10            // Create an executor and execute the protocol.
11            Executor executor = new Executor(programConfiguration);
12            return executor.compileProgramToDmf();
13        } catch (DmfException e) {
14            // If there is a DMF exception, return a ExecutionResult
15            // with the error message.
16            return new ExecutionResult(e.getMessage());
17        }
18    }
19 }
```

Upon receiving a POST request at `/api/compile`, the controller processes the input data through the `ProgramExecutionService` class. If the execution result contains an error, it will return the result with the HTTP status code `400 Bad Request`, indicating that an error occurred while processing the data. If the execution is successful, it will return the execution result with the HTTP status code `200 OK`.

Benefits of this approach:

- **Single Access Point:** This ensures a simple interface. where the flow of each process is well defined. Each process is processed one at a time, letting Spring handle the queuing of requests.
- **Synchronous Processing:** The request-response cycle is maintained until the processing of a request is complete, ensuring the client awaits the execution result directly. This eliminates the need for complex coordination and state management associated with distributed systems.

By utilizing Spring Boot to facilitate a REST API, we ensure a loosely coupled and flexible backend, which can handle the requests received from the frontend.

5.2.2 Input Reader

The ‘Input Reader’ component seen in Figure 4.8 refers to the process of reading and mapping JSON data into internal models, rather than a specific class or implementation. This process involves reading JSON input, mapping it to temporary JSON specific Data Transfer Object (DTO) classes, implementing custom deserializers, and finally converting these DTOs into internal models that the executor can understand.

Initial Access Point

The initial access point for reading JSON data, is the `JsonModelLoader` class. This class uses Jackson to deserialize JSON data into Java objects. Jackson is a popular and widely used library for processing JSON in Java, providing powerful features for deserialization, which we will be utilizing in this project.

Suiting DTO models, mirroring the structure of each level in the received JSON object (see Listing 4.1 for reference), have been created to facilitate this mapping.

The `JsonProgramConfiguration` class represents the overall structure of the program’s JSON configuration, and includes the initial state of the DMF platform and a list of actions in the protocol. The Jackson annotation `JsonProperty` seen in Listing 5.13, guides Jackson as to which parts of the JSON object it should map to which models.

Listing 5.13: `JsonProgramConfiguration.java`

```
1 @JsonIgnoreProperties(ignoreUnknown = true)
2 public class JsonProgramConfiguration {
3     @JsonProperty("dmf_configuration")
4     private JsonDmfPlatformState dmfPlatformState;
5
6     @JsonProperty("program_actions")
7     private List<JsonActionBase> programActions;
8 }
```

Reading and parsing actions into DTO models

We already discussed `JsonProgramConfiguration` as an example of a DTO model. We will now zoom in on the process of mapping the JSON representation of actions into suitable DTO models using Jackson.

For each action type (e.g., `Input`, `Move`, `Merge`, etc.), a suitable model has been created, containing the relevant attributes for each action. Common for all of these is that they are all sub-types of the parent class `JsonActionBase`, which provides Jackson with the necessary information to determine how to distinguish between actions.

In Listing 5.14 it is seen, how the Jackson annotations `@JsonTypeInfo` and `@JsonSubTypes` are used to guide the deserialization process by specifying how Jackson should determine the type of each action based on the `action` property in the JSON data.

Listing 5.14: `JsonActionBase.java`

```
1 @JsonTypeInfo(  
2     use = JsonTypeInfo.Id.NAME,  
3     include = JsonTypeInfo.As.PROPERTY,  
4     property = "action") //Property used to determine action type.  
5  
6 // Inform Jackson how to map each action to internal models,  
7 // based on the "action" property defined above.  
8 @JsonSubTypes({  
9     @JsonSubTypes.Type(value = JsonInputAction.class, name = "input"),  
10    @JsonSubTypes.Type(value = JsonOutputAction.class, name = "output"),  
11    @JsonSubTypes.Type(value = JsonMoveAction.class, name = "move"),  
12    /* other action types..*/  
13 })  
14  
15 @Getter  
16 public class JsonActionBase extends DtoActionBase {  
17     public JsonActionBase(String id) {  
18         super(id);  
19     }  
20 }
```

Custom Deserializers for Input Parameters

To ensure the validity of the input data, custom Jackson deserializers for each data-type contained in the received JSON object have been implemented. These deserializers throw specific exceptions, when invalid data is encountered. For instance, the `StrictNonEmptyIntegerDeserializer` ensures that the corresponding field in the JSON object is not empty, as well as ensuring that the value can be parsed as an integer. Similar deserializers have been created for the `String` and `Decimal` types, throwing a `DmfInvalidInputException` if any invalid data is found.

Mapping DTOs to Internal models

After parsing the JSON data into DTOs that Java can understand, the next step is converting these into internal models, that the executor can understand. This distinct separation between JSON DTOs and internal models allows us to completely decouple the data reading process from the execution logic, making it easier to support different data formats, such as XML, in the future, if needed.

In the following, we will focus on the conversion of Actions into internal models, as this process is generally more complex and interesting due to the use of polymorphism to handle the different types of actions. It should be noted that a similar process is followed for mapping the DMF platform configuration into internal models.

IActionMapper Interface

To correctly map a DTO object to an internal action, each DTO class has a corresponding mapper, that handles this conversion. The `IActionMapper` interface seen in Listing 5.15 defines the contract for these mappers, ensuring a consistent approach to mapping and reference resolution.

Listing 5.15: IActionMapper.java

```

1 public interface IActionMapper<T extends DtoActionBase,
2     U extends ActionBase> {
3
4     U mapToInternalModel(T dtoModel);
5     void resolveReferences(T dtoModel, Map<String, U> internalModelMap,
6         Map<String, Droplet> dropletMap);
7 }

```

For example, the `JsonMoveActionMapper`, shown in Listing 5.16, converts a `JsonMoveAction` DTO into a `MoveAction`, which is the internal representation of a move action that the executor can work with.

Listing 5.16: JsonMoveActionMapper.java

```

1 public class JsonMoveActionMapper implements IActionMapper<
2     JsonMoveAction, MoveAction> {
3     @Override
4     public MoveAction mapToInternalModel(JsonMoveAction dtoModel) {
5         return new MoveAction(
6             dtoModel.getId(),
7             dtoModel.getPosX(),
8             dtoModel.getPosY()
9         );
10    }
11 }

```

```

12     @Override
13     public void resolveReferences(
14         JsonMoveAction dtoModel,
15         Map<String, MoveAction> internalModelMap,
16         Map<String, Droplet> dropletMap) {
17
18         MoveAction moveAction = internalModelMap.get(dtoModel.getId());
19         moveAction.setNextAction(
20             internalModelMap.get(dtoModel.getNextActionId()));
21         moveAction.setDroplet(dropletMap.get(dtoModel.getDropletId()));
22     }
23 }

```

Although the mapping process may seem redundant at first glance, especially for simple one-to-one mappings like in the `JsonMoveActionMapper`, it becomes advantageous for more complex actions. For instance, in the `JsonMixActionMapper`, the mix action is mapped into a sequence of internal actions (an `ActionQueue`), which involves multiple `MoveAction` instances that together accurately represents the mixing process. This approach allows us to decompose more complex actions into smaller, already existing actions in the backend. This makes it easier to extend the system with additional action types and functionalities in the future, if need be.

Factory for Action Mappers

To streamline the conversion process of actions, a factory pattern has been employed (see Listing 5.17), to manage the different action mappers. The `JsonActionMapperFactory` provides an easy way to dynamically obtain the appropriate mapper, for each action type.

Listing 5.17: `JsonActionMapperFactory.java`

```

1 public class JsonActionMapperFactory {
2     private static final Map<Class<? extends DtoActionBase>,
3         IActionMapper<?, ?>> mappers = new HashMap<>();
4
5     static {
6         mappers.put(JsonInputAction.class, new JsonInputActionMapper());
7         mappers.put(JsonOutputAction.class, new JsonOutputActionMapper());
8         mappers.put(JsonMoveAction.class, new JsonMoveActionMapper());
9         /* Other mappers... */
10    }
11
12    public static <T extends DtoActionBase> IActionMapper<T, ?>
13        getMapper(Class<T> dtoClass) {
14        return (IActionMapper<T, ?>) mappers.get(dtoClass);
15    }
16 }

```

Resolving References

The `ResolveReferences` method found in the `IActionMapper` interface, may initially seem confusing, but it is crucial for establishing the correct relationships between components in the internal model. In the DTO models, references such as `nextAction` are represented as strings containing the ID of the next action. These IDs need to be converted into direct object references in the internal models.

The conversion should thus be done in two passes: first, all actions are created; second, all references are resolved. This ensures that all actions has been correctly instantiated, before the references are set.

Program Configuration

After successfully mapping the JSON DTO models, that is, all actions as well as the DMF platform configuration, into internal models that are understood by the executor, they are combined into an instance of `ProgramConfiguration`. This model encapsulates all the necessary data needed for executing the protocol and houses various data structures derived from the converted JSON data, which are used during execution. A more detailed description of these follows in the coming sections.

5.2.3 Droplets

In this section, we will provide an overview of how a droplet is internally represented within the compiler. All actions presented in this thesis, involves the manipulation of one or more droplets, and it is therefore important to have a well-defined `Droplet`-class.

The class representing a droplet includes several parameters and methods that is important for correctly manipulating droplets on the DMF platform. The general structure of the `Droplet` class can be seen in Figure 5.2, where only the most relevant methods is shown.

Calculating Droplet Diameter

The diameter of a droplet is calculated using its volume, as well as the height between the DMF platform and the glass, which is currently set to 0.5 millimeters. Given the small height, the droplet's shape can be approximated as a cylinder, as it is touching the electrodes below and the glass on top at the same time. The formula for the diameter d of a cylinder, given the Volume V and the height h can be calculated with the formula $d = 2\sqrt{\frac{V}{\pi h}}$.

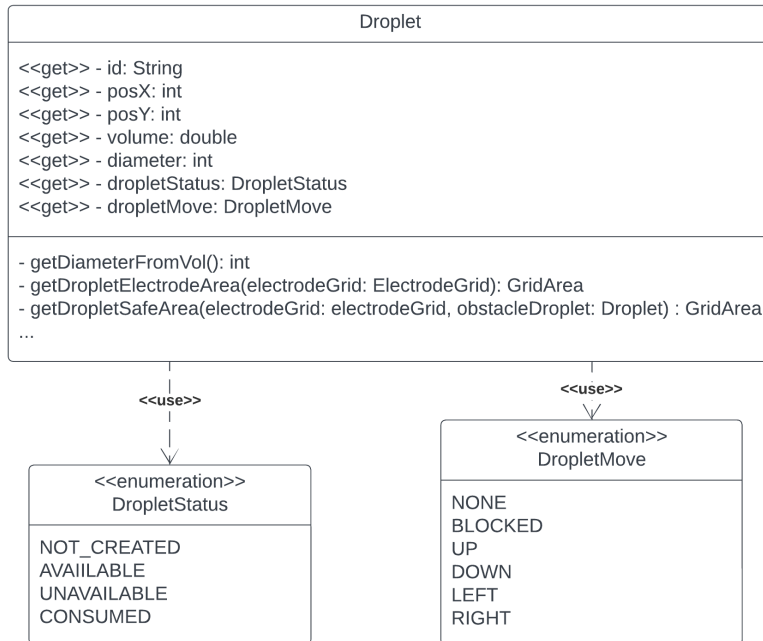


Figure 5.2: A class diagram depicting the most important attributes and methods for the **Droplet** class.

Important methods

While the **Droplet** class houses various helper-methods used during execution, there is especially two methods, which are important for moving droplets on the platform:

1. **getDropletSafeArea** - This method, provided another **Droplet** instance which we call an **ObstacleDroplet**, calculates the electrodes on the DMF platform which the current droplet cannot be located at, while maintaining a safe distance of at least one electrode between the two droplets. This distance is necessary to avoid unintended collisions between droplets. The size of the area is calculated using the ceiled diameter of the droplet (in electrodes) + 1 electrode, to ensure a proper distance between the droplets of at least one electrode. The area returned by the method conforms to a rectangle, simplifying the logic for calculating the safe area.

2. **getDropletElectrodeArea** - This method returns the area (electrodes) beneath the droplet, that should be activated, to move the droplet around on the platform. The returned electrodes form a rectangle, simplifying the logic for moving or otherwise manipulating the droplets. The size of this area is determined using the floored diameter of the droplet (in electrodes), as droplets typically conform to the shape of the set of returned electrodes (i.e. rectangular), resulting in a slightly smaller effective diameter.

Since the safe area is always calculated from the actual diameter (which is the largest a droplet may be), it ensures that the calculated area is always equal to or larger than necessary, but never smaller, thus preventing any collisions.

An illustration of how these methods work in practice is presented in Figure 5.3.

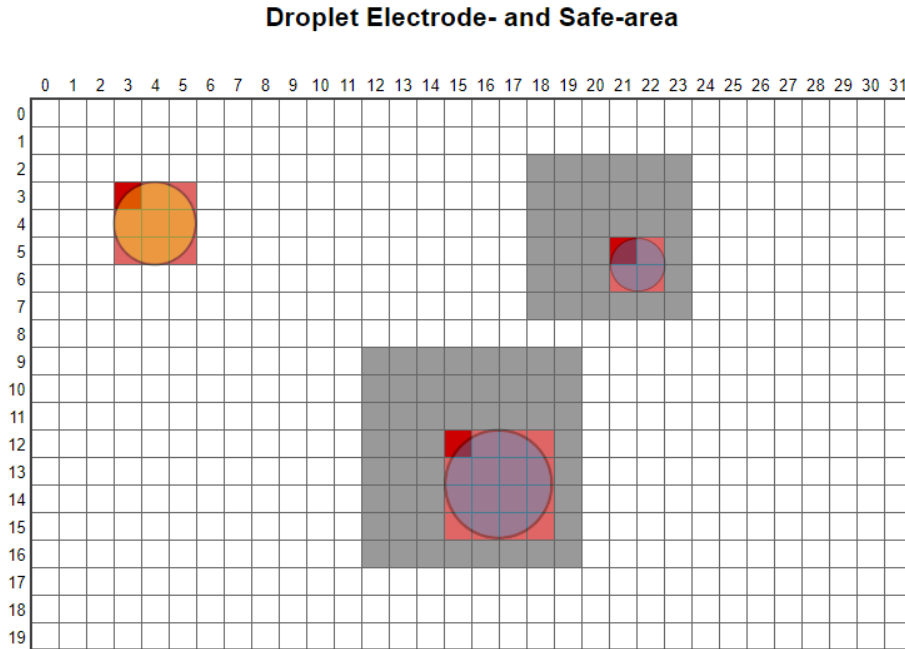


Figure 5.3: An illustration showing the area returned by `getDropletSafeArea` and `getDropletElectrodeArea`. The red squares indicate the electrodes used to move the droplet (`getDropletElectrodeArea`) while the gray area indicate the area occupied by the blue obstacle droplets (`getDropletSafeArea`).

Positional Considerations

It is important to note that the position (`posX` and `posY`) assigned to the droplet, is not marking the center of the droplet, but rather the top-left corner of the rectangle bounding the circumference of the droplet. In Figure 5.3, these locations are marked as a dark-red square, and the yellow droplet consequently has the position (3,3) in the provided example.

It should be noted, that droplets are moved by a set of electrodes, which together form the shape of a rectangle. This simplification makes the logic for manipulating droplets much simpler, and does not introduce any significant drawbacks in terms of precision and control.

5.2.4 Actions

In this section, we will describe the internal actions used by the executor. These actions represents core operations that can be performed by a droplet. While most actions have corresponding blocks in the frontend, their functionality is not always mapped one-to-one. It is important to clarify, that in this section, whenever we refer to actions, we are discussing the internal models used by the executor, not to be confused with the temporary DTO action models mentioned in Section 5.2.2, nor the blocks mentioned in Section 4.4.3 or the ‘Program Actions’ mentioned in Section 4.5.1.

For example, the `StoreAction` does not move the droplet to a specific location, even though a location is present in the store block described in Section 4.4.3. Instead, the block is decomposed into a `ActionQueue` containing a `MoveAction` and a `StoreAction`. The `StoreAction` is thus only responsible for storing the droplet for a certain amount of ticks, making it unavailable for other actions during this period.

All actions extend the abstract class `ActionBase`. This class defines the main methods that all actions must implement. An outline of the `ActionBase` class as well as specific implementations of said class can be seen in Figure 5.4, though only showing the relevant parameters and methods relevant for this section.

The `ActionTickResult` object being returned in the `ExecuteTick` method as seen in Figure 5.4, encapsulates the result of executing a single tick. It contains a list of commands to be executed on the DMF platform (e.g. setting and clearing of electrodes), as well as a flag indicating whether anything actually happened during the tick. While the latter may seem redundant at first glance, it is actually necessary for facilitating the functionality of the `StoreAction`, where multiple ticks may need to be performed, without the DMF Platform changing

its state. The ability to check if anything happened in a tick is especially important in the context of checking whether a deadlock has been encountered, which we will discuss in further details in Section 5.2.9.

Important methods that each action must implement:

- **dropletsRequiredForExecution:** This method is mainly used by the schedule, to determine if an action can be started, by ensuring all required droplets are available. Scheduling of droplets is further discussed in Section 5.2.7.
- **dropletsProducedByExecution:** This method identifies the droplets that are generated by the action. This is particularly relevant for actions like `InputAction`, `SplitAction` and `MergeAction`, where new droplets are created as a result of the action.
- **beforeExecution:** The `beforeExecutionAction` is called initially, before the action is ticked for the first time. This typically involves marking the action as ‘IN_PROGRESS’ and marking the required droplets as ‘UNAVAILABLE’, ensuring that the droplets are not used by multiple actions simultaneously. Additionally, the `ProgramConfiguration` object is passed in, providing access to the state of the platform.
- **executeTick:** This method performs the actual tick of the action and returns the result in the form of an `ActionTickResult`. This involves the setting or clearing electrodes if the action is manipulating droplets on the platform. The `ActionTickResult` can also be empty if the action for any reason is not able to progress during a tick. The `ProgramConfiguration` object is passed in, providing access to the state of the platform, most notably references to all droplets on the platform.
- **afterExecution:** This method runs once after the action has been marked as completed. This usually involves marking any droplets not consumed by the action as ‘AVAILABLE’, making them ready for any subsequent actions in the following tick.
- **verifyProperties:** This method verifies that the action’s properties are valid, in respect to the ‘`ProgramConfiguration`’. An example of this could be ensuring that all positions defined for the action, is within the bounds of the platform specified in the `ProgramConfiguration` object.

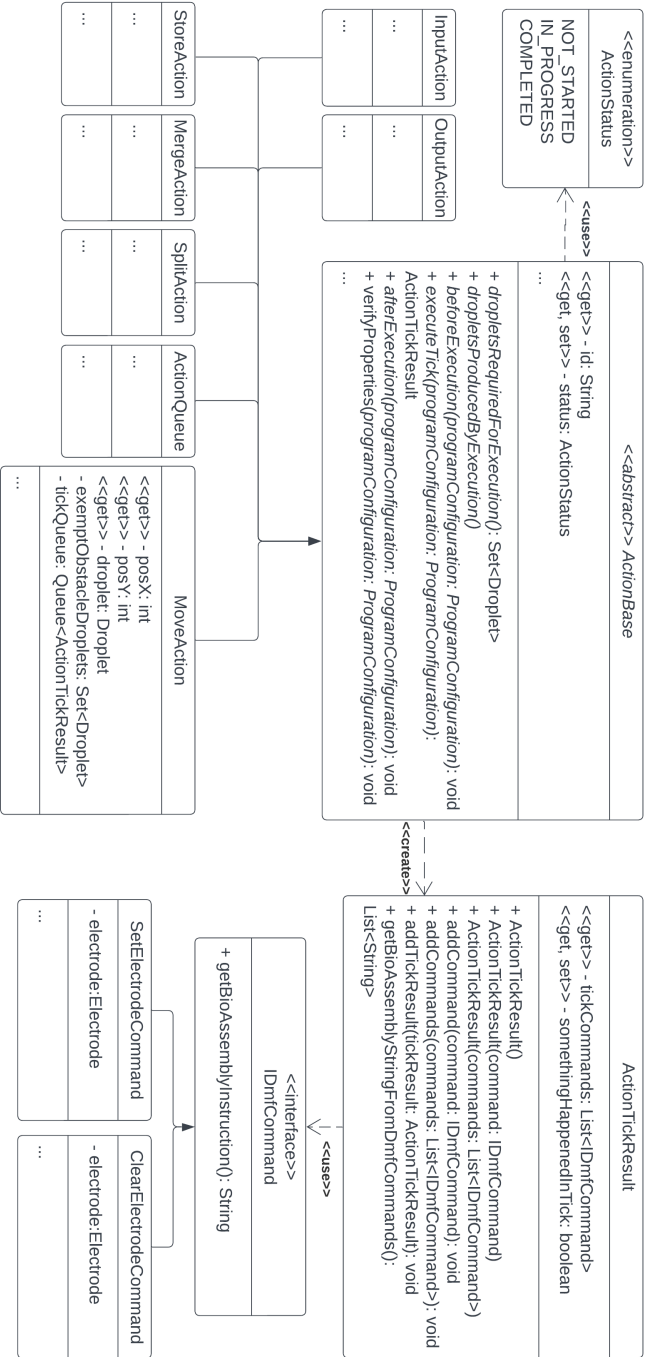


Figure 5.4: A class diagram depicting the relevant attributes and methods for the `ActionBase` class, as well as relevant classes implementing `ActionBase`.

We will now describe each action in greater detail, explaining what they do and how they work.

- **InputAction** The **InputAction** is responsible for placing a droplet at a specified position on the DMF Platform. This is achieved, by activating the electrodes at the specified position, whereafter the droplet is marked as **AVAILABLE**. The actual creation of a physical droplet on the DMF Platform is omitted for now, leaving this as an obvious extension to the system, namely implementing support for sensors and actuators. This is, however, beyond the scope of this thesis.
- **OutputAction** The **OutputAction** marks the droplet as **UNAVAILABLE**, effectively ensuring that the droplet cannot be obtained by any subsequent actions. The droplet remains visible to the platform, such that collisions with other droplets are avoided.
- **MoveAction** The **MoveAction** moves the droplet to a specified position. This involves determining which way to move the droplet each tick, ensuring that no unintended collisions are performed, activating and deactivating electrodes as necessary, to move the droplet. The action often spans multiple ticks, until the droplet reaches the target position.
- **StoreAction** The **StoreAction** marks a droplet as **UNAVAILABLE** for a specified number of ticks, making it unavailable to other actions. This action is responsible for only the storage duration, while the responsibility of initially moving the droplet is given to a preceding **MoveAction**.
- **ActionQueue** The **ActionQueue** represents a sequence of actions, that are to be executed in order, and each action in the queue must be completed, before the next one starts. The action ensures that all droplets required to run any of the actions contained in the queue is available, before the action is started. The **Action** is marked as **COMPLETED**, once all internal actions has been completed.
- **MergeAction** The **MergeAction** is responsible for moving two droplets to a common target position. This process involves gradually moving the droplets closer together and allowing them to collide only when one of them reaches the target position, which ensures that no premature merging takes place. Upon completion, the newly created droplet is marked as available, while the two origin droplets are marked as consumed, effectively stating, that they are no longer present on the platform.
- **SplitAction** The **SplitAction** splits a droplet into two equally sized smaller droplets, by splitting the droplet in half in either the horizontal or vertical direction, prioritizing the horizontal splits over vertical if possible.

This is achieved by manipulating the electrodes in such a way, over three consecutive ticks, that a gap of three electrodes becomes present between each part of the split droplet. In the following we will focus on the case of horizontal splits, but it should be known, that a similar process is implemented for vertical splits. Initially, a row of electrodes is cleared in the bottom of the droplet, to prepare for the splitting of droplets. Additionally, if the droplet spans an even amount of electrodes, an extra column of electrodes is set to the right of the droplet, to ensure that the droplet is spanning a uneven amount of electrodes, allowing for an even split. In the next two consecutive ticks, the droplet is pulled into two separate droplets, as depicted in Figure 5.5. When the split has been completed, it is followed by two `MoveActions`, each moving a droplet to their individual target position.

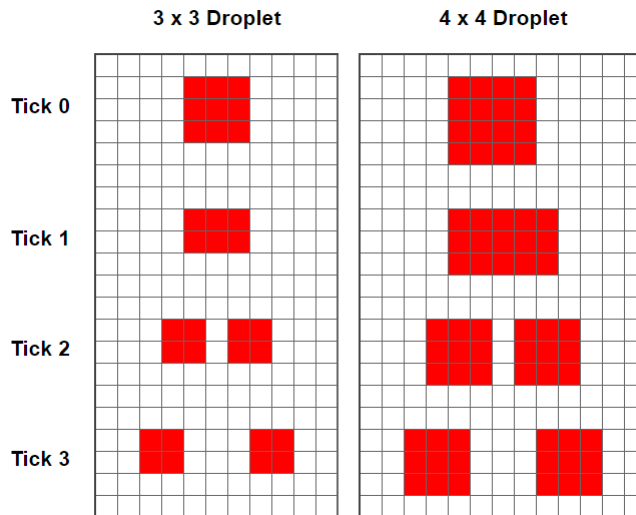


Figure 5.5: Illustration of 2 droplets splitting in the horizontal direction tick by tick. The red squares indicate electrodes with a voltage. Droplets have been omitted for simplicity.

MoveAction - A Concrete Example

We will now delve deeper into the details of the `MoveAction`, which encapsulates important aspects such as pathfinding, collision avoidance and stepwise execution. Understanding these aspects is important for the understanding of the `MoveAction` along with the rest of the actions. The concrete implementation details of the remaining actions follow similar principles and can be found in the source code provided in Section 1.5.

The `MoveAction` is responsible for moving a droplet to a specified position on the electrode grid of the DMF platform. This involves determining the optimal direction, if any, for a droplet to move in a given tick, to avoid undesired collisions and to move in an optimal path. We will discuss the topics of determining valid positions a droplet can be moved to in Section 5.2.5, as well as details about pathfinding in Section 5.2.6.

Moving a droplet one electrode, takes two clock-ticks on the DMF platform. In the initial tick, the row/column in front of the droplet in the direction of movement is set, while the electrodes behind the droplet is cleared in the following tick. To handle this, the `MoveAction` must be able to know that it is in the middle of moving the droplet in the next tick. This is achieved, by, in the first tick, calculating how to perform both consecutive ticks, and store the results in the form of `ActionTickResult`-objects in the `tickQueue`. Whenever the action is ticked, the `tickQueue` is checked. If anything is present in the queue, the `ActionTickResult` will be popped and returned immediately, without any further computations.

This principle of using a `tickQueue` to manage multi-tick operations is also employed in other actions, such as the `MergeAction` and `SplitAction`.

5.2.5 Electrode Grid

The provided `ProgramConfiguration` contains a `PlatformInformation` object which, among other data, holds the list of electrodes. However, to accurately represent the grid of electrodes, as that of the actual DMF platform, we will convert the list of electrodes to a 2D-array of electrodes. This is achieved in the `ElectrodeGridFactory` class.

Assuming that all electrodes are of the same size in the grid we can retrieve the electrode size from the first electrode in the list. The size of the platform (entire grid) can be retrieved from the same `PlatformInformation` object as the list of electrodes. By dividing the size of the grid with the electrode size in both the x and y dimension we are able to calculate `xMax` and `yMax` which is the number of electrodes in each dimension. This allows us to construct the grid of electrodes in the correct dimensions as shown in Listing 5.18. This also means that we will not be supporting if the platform size is not directly divisible by the electrode size, which could happen if electrodes are of different size or if there are missing electrodes or gaps.

Listing 5.18: Creation of the ElectrodeGrid

```

1 public static ElectrodeGrid getElectrodeGrid
2   (ProgramConfiguration programConfiguration) {
3
4   // Get list of electrodes
5   List<Electrode> electrodes = programConfiguration
6     .getPlatformInformation().getElectrodes();
7
8   // Get size of electrodes (assuming that all electrodes are the
9   // same size as the first)
10  int electrodeSizeX = electrodes.get(0).getSizeX();
11  int electrodeSizeY = electrodes.get(0).getSizeY();
12
13  // Get size of platform
14  int platformSizeX = programConfiguration.getPlatformInformation()
15    .getSizeX();
16  int platformSizeY = programConfiguration.getPlatformInformation()
17    .getSizeY();
18
19  // We will not support if platform size is not directly divisible
20  // by electrode size.
21  if(platformSizeX % electrodeSizeX != 0 || platformSizeY %
22    electrodeSizeY != 0) {
23    throw new IllegalArgumentException("Platform size not directly
24    divisible by electrode size");
25  }
26
27  int xMax = (platformSizeX / electrodeSizeX);
28  int yMax = (platformSizeY / electrodeSizeY);
29
30  // Grid will be Electrode[x][y]
31  Electrode[][] grid = new Electrode[xMax][yMax];
32  for (int y = 0; y < yMax; y++) {
33    for (int x = 0; x < xMax; x++) {
34      int currentElectrodeIndex = y * xMax + x;
35      grid[x][y] = electrodes.get(currentElectrodeIndex);
36    }
37  }
38  return new ElectrodeGrid(grid);
39 }

```

The class diagram in Figure 5.6 illustrates the structure and relationships of the `ElectrodeGrid`, `GridArea`, and related command classes. The `ElectrodeGrid` class manages the grid of electrodes, providing methods for removing and checking the bounds of electrodes. The `GridArea` class defines a specific area within the grid. The `SetElectrodeCommand` and `ClearElectrodeCommand` classes, which implement the `IDmfCommand` interface, provide the logic for generating BioAssembly instructions for setting and clearing individual electrodes.

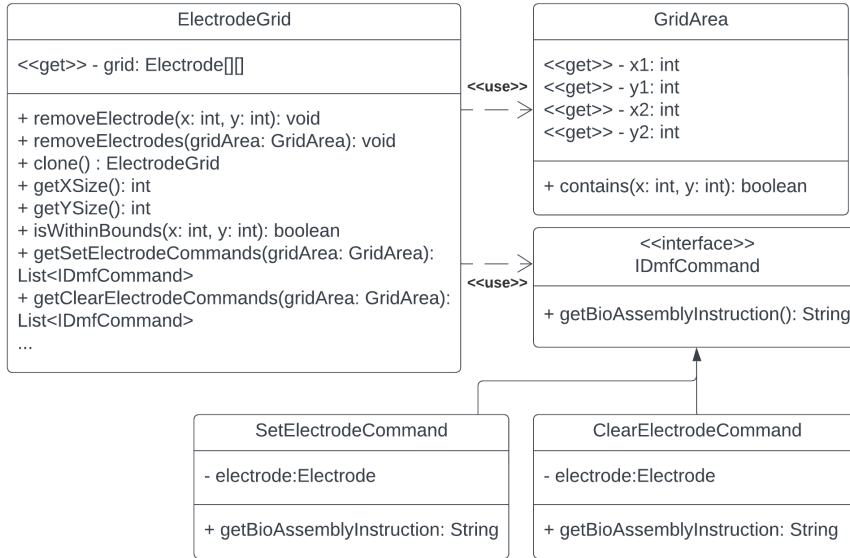


Figure 5.6: A general class diagram, showing the most important attributes and methods on the **ElectrodeGrid** class.

Calculating the AvailableElectrodeGrid

From the **ElectrodeGrid** we can calculate the **AvailableElectrodeGrid**, which is the grid of available electrodes for a given droplet. We will do so by removing electrodes that are not accessible to the droplet that we are currently moving. The **AvailableElectrodeGrid** is calculated in the **ElectrodeGridFactory** class as well, in the method `getAvailableElectrodeGrid` which takes three parameters:

- **electrodeGrid** - the **ElectrodeGrid** object from which electrodes are to be removed.
- **activeDroplet** - the **Droplet** object for which we are calculating the available electrodes.
- **obstacleDroplets** - a list of **Droplet** objects representing other droplets on the grid which will be seen as obstacles for the **activeDroplet**.

When calculating the **AvailableElectrodeGrid** we have to remember that we are representing droplets as squares, with their position being the upper-left corner. Because of this we will, based on the droplet's size, be removing electrodes from the electrode grid at the top and left side of the obstacle droplets,

along with a perimeter of one electrode around the obstacle droplets as the safe area. This is done to avoid unintended droplet collisions and breach of safe areas. We will also be removing electrodes from the bottom and the right border of the electrode grid to avoid the droplet spilling over the edges. An illustration of which electrodes are removed from the `ElectrodeGrid` to calculate the `AvailableElectrodeGrid` for one `ActiveDroplet` with two `ObstacleDroplets` are shown in Figure 5.7.

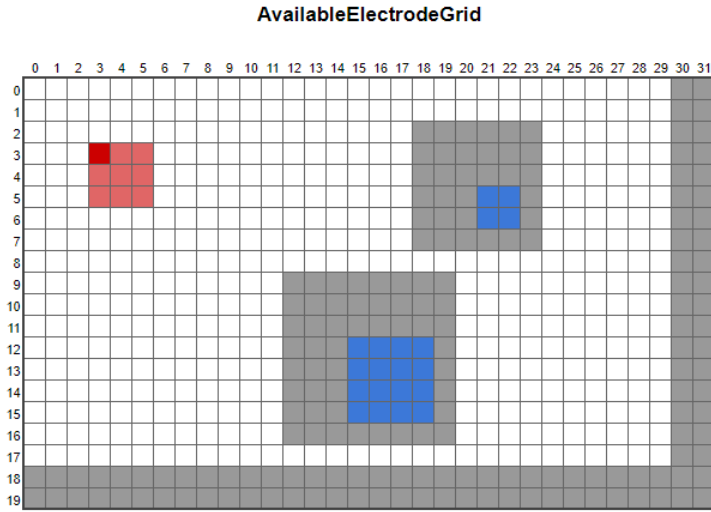


Figure 5.7: AvailableElectrodeGrid example

In Figure 5.7 the red square is representing the `ActiveDroplet` with the dark red electrode being the current position of that droplet. The blue squares represent the `ObstacleDroplets` and the grey areas represent removed electrodes that are inaccessible to the `ActiveDroplet`. One very important thing to note is that we are only using the size of the droplet to calculate the available electrodes. Meaning, that after the `AvailableElectrodeGrid` have been calculated the droplet can be thought of as a single electrode, because we already removed the electrodes that are inaccessible due to the size of the `ActiveDroplet`. This means that our red droplet can in fact pass between the blue droplets, because we only have to consider the dark red electrode as being the space the `ActiveDroplet` needs to move around. Put another way, the `ActiveDroplet` can be thought of as the size of a single electrode in the `AvailableElectrodeGrid` calculated for that specific droplet, which means that the red droplet can be placed on any of the white electrodes, as these are available.

5.2.6 A* - A Path-Finding Algorithm

To ensure optimal paths of the droplets, we have implemented a path-finding algorithm in the form of A-star (A*) [26]. The A-star is a very popular algorithm for finding a path from a starting node to an end node in a graph, as it can do so very efficiently. In the following sections we will present the basics of the A-star algorithm. This will be followed by details specific to our implementation, hereunder the heuristic that was used in this project and an explanation of our addition of finding the closest node if the goal node is inaccessible.

A-star basics

The A-star algorithm starts at a given node, often called the start node and from there it will find a path to the goal node. Initially however, the algorithm will only know about the start node. The graph search version of A-star algorithm, which we have used in this project, maintains two lists: an open list and a closed list. The open list contains the nodes that will need to be evaluated, whereas the closed list contains nodes that have already been visited.

The A-star algorithm contains a loop which will continue until a goal node have been reached or the open list is empty, i.e. until there are no more nodes to be explored. In the loop the algorithm will explore the nodes with the lowest f -value first. The value $f(n)$ for a node n is the sum of the value of the heuristic function to reach that node, $h(n)$, and the cumulative cost to reach that node, $g(n)$, as expressed by Equation 5.1.

$$f(n) = h(n) + g(n) \quad (5.1)$$

The algorithm will then check if the node being explored is the goal, and if so, it will end. Otherwise, it will continue the search, maintaining both the open and closed lists. From the current node the algorithm will find neighboring nodes, which are nodes that can be accessed directly from the current node. Neighboring nodes not in the open list will be added, except for those that have already been expanded and are present in the closed list. Commonly, the A-star algorithm will also update the path if a better one is found, i.e. a path that will result in a lower f -value.

Our A-star implementation

Now that we have introduced the basics of the A-star algorithm we will dive into greater detail of our implementation. In our solution, the algorithm is used to calculate the optimal path for a droplet to move, from the initial position to the goal position. However, because we have multiple droplets which will be moving at the same time, we will not be using the full path, as calculated by the A-star algorithm, directly. Instead, we will only be using the first move, defined by the enums UP, DOWN, LEFT or RIGHT, for each moving droplet in every tick.

Our implementation of the A-star algorithm works on a grid of electrodes, more specifically `AvailableElectrodeGrid`, which is the grid of available electrodes for the droplet being moved as described in Section 5.2.5. This ensures that all electrodes in the grid are in fact available for the current droplet that we are calculating the path for. For the remainder of this section we will be using node and electrode interchangeably.

Because a droplet is only allowed to move horizontally or vertically, we have set the step cost to be 1, as a step will always be of the same size since no diagonal movements in the grid are allowed.

For the heuristic function we have used the Manhattan distance, which is calculated as the sum of absolute differences in the x and y dimensions. The formula for calculating Manhattan distance is presented in Equation 5.2.

$$dist = |x_1 - x_2| + |y_1 - y_2| \quad (5.2)$$

Using the Manhattan distance as the heuristic function is great because it can be considered admissible for our use case, where droplets are moving in a grid with no diagonal moves allowed. A heuristic function is admissible if it never overestimate the cost of reaching the goal state [27]. If $h(n)$ is the estimated cost to reach the goal node and $h^*(n)$ is the actual cost, the formal definition of an admissible heuristic is as defined in Equation 5.3.

$$h(n) \leq h^*(n) \quad (5.3)$$

If the target position of a droplet's movement is blocked, the basic A-star implementation will not be able to find a path. Therefore, we have modified our solution to handle this specific case. We do so by allowing the droplet to accept the closest possible node, as the new goal state if it cannot reach the actual goal. In practice this is achieved by keeping track of the node with the best heuristic (lowest value). This means that if the A-star algorithm terminates

before reaching the goal node, because the open list is empty, we will accept the closest node as the new goal, allowing the droplet to still be able to move towards the actual goal.

While this is great because it allows droplets to move closer towards their actual goal when it is temporarily blocked, it will not solve the problem of the goal being permanently blocked. In this case the droplet move will be impossible and we will attempt to resolve the deadlock as described in Section 5.2.9.

5.2.7 Scheduling of Droplets

Scheduling is a crucial part of the compilation process. Proper action scheduling can drastically optimize the execution time, and thus the resources used by the DMF platform, by allowing multiple droplets to move concurrently, whenever possible. The `Schedule` class seen in the general class diagram shown in Figure 5.1 manages this process, ensuring that actions are executed in the correct order, and only when all the required droplets are available.

Construction of the Schedule

The `Schedule` class is created from a list of actions, sorted in the order of which they should be executed. During the creation process, all droplets involved in the protocol, are mapped to a queue of all actions which require the aforementioned droplet to be executed. The actions in each queue are ordered in respect to the order of which they appeared in the original list of actions, and a certain action may appear in multiple queues, if multiple droplets is required to execute the action.

When a `Schedule` object is instantiated, a map (`dropletActions`) that associates each droplet with a queue of actions is initialized. During the population of these queues, the schedule verifies that each droplet is only produced by exactly one action, and that no actions require droplets that has not yet been produced by any preceding actions. If any errors is detected in the protocol, such as attempting to move a droplet with a `MoveAction` before the droplet has been produced, e.g. by a `InputAction`, an exception is thrown, terminating the execution prematurely, providing a descriptive error message to be displayed in the frontend.

The mapping and validation process is achieved by doing two consecutive checks for each action, as seen in Listing 5.19. Initially, all droplets being produced by the action, is added to `dropletActions` with an empty queue of actions. Secondly, the action is added to the respective queues of all droplets that are required for the action to be executed. Input actions need to be explicitly added

to their respective droplet queues because they, as the only action, do not require a droplet to be available for the action to be executed.

Listing 5.19: Constructor of Schedule.java. Some details have been omitted for simplicity.

```

1 public Schedule(List<ActionBase> actions) {
2     dropletActions = new HashMap<>();
3
4     actions.forEach(action -> {
5         Set<Droplet> producedByExecution =
6             action.dropletsProducedByExecution();
7         for (Droplet droplet : producedByExecution) {
8             if (dropletActions.containsKey(droplet.getID())) {
9                 /* DropletID is produced by multiple actions */
10                throw new DmfInvalidInputException( /* ... */ );
11            }
12            // Add a Queue to dropletActions
13            dropletActions.put(droplet.getID(), new LinkedList<>());
14
15            // Input actions does not require the droplet to be
16            // executed, and thus needs to be added explicitly.
17            if (action instanceof InputAction) {
18                dropletActions.get(droplet.getID()).add(action);
19            }
20        }
21
22        Set<Droplet> dropletsRequiredForExecution =
23            action.dropletsRequiredForExecution();
24        if (dropletsRequiredForExecution == null) return;
25        for (Droplet droplet : dropletsRequiredForExecution) {
26            if (!dropletActions.containsKey(droplet.getID())) {
27                /* DropletID is required before being produced */
28                throw new DmfInvalidInputException( /* ... */ );
29            }
30            dropletActions.get(droplet.getID()).add(action);
31        }
32    });
33 }

```

Getting Actions to Be Ticked

To request a list of all actions that can be executed in the current tick, the public method `getActionsToBeTicked` can be called. This method starts by updating the schedule, by removing all actions from the queues, that are marked as `COMPLETED` (e.g. if an action was completed in the previous tick). It then checks each queue, corresponding to the respective droplet, if the next action in the queue can be executed.

An action can be executed only when all queues for all droplets required by the action have the action as their next action. This ensures that an action can only be performed once all droplets involved in said action, has completed all their preceding actions, maintaining the integrity of the original protocol provided by the user.

The method `getActionsToBeTicked` iterates through each droplet's action queue, and performs the following checks, to determine if an action can be ticked:

1. **Check Droplet Availability:** If the action has not yet started (e.g. being marked as `NOT_STARTED`), it is a requirement, that all droplets required by the action is marked as `AVAILABLE`. This is necessary, since a droplet is not necessarily produced, by the the action received in the queue. An example of this, could be a droplet produced by a `Merge` or `Split` action, resulting in said droplet being marked as `NOT_CREATED`. By ensuring all actions required for an action is `AVAILABLE` before initially starting the action, it is guaranteed, that a droplet has been produced (`AVAILABLE`) prior to starting the action.
2. **Check if action is the next action for required droplets:** When all required droplets are `AVAILABLE`, or the action is already marked as `IN_PROGRESS`, the action is deemed tickable if all queues corresponding to a droplet which is required by the action have the action as their next action in the queue. Please note, that since the droplets may already be marked as `UNAVAILABLE` (e.g. when initially starting the action in a previous tick), it is not a requirement that all droplets are `AVAILABLE`, when the action is marked as `IN_PROGRESS`. It is here assumed, that all droplets required by the action, is already reserved for said action.

Example of Scheduling Execution

To better illustrate the scheduling process and how the `Schedule` class manages actions internally, we will walk through the state of the schedule during the execution of a small example protocol. The protocol involves a sequence of actions, consisting of `Input`, `Move`, `Merge` and `Output` operations for droplets on the DMF platform.

Figure 5.8 shows the aforementioned protocol in block-form. The protocol starts by inputting the droplets D1 and D2 onto the platform. D2 is then moved, whereafter D1 and D2 is merged into D3. Finally, D3 is moved, and subsequently outputted.

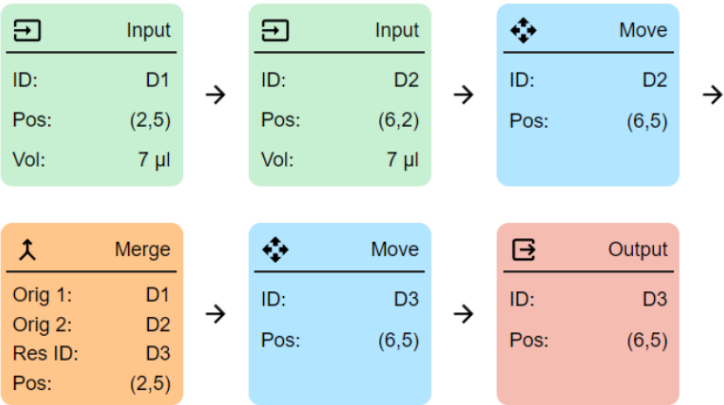


Figure 5.8: Example protocol: Input two droplets (D1 and D2), move D2, merge D1 and D2 into D3, move D3, and output D3.

Figure 5.9 shows the progression of the example protocol for each tick. The figure shows the status of the electrode grid (to the left), as well as a visual representation of the schedule, and the status of each droplet (to the right). Each sub-figure displays the state of the electrode grid at a specific tick. Droplets are represented by colored squares, with the different colors indicating different droplets. Next to the grid-representation, a table tracking the status of each droplet (e.g. NOT_CREATED, AVAILABLE, UNAVAILABLE, CONSUMED), and the actions in their respective queues.

Arrows are used when the sub-figure spans multiple ticks to indicate that the droplet is moving during these ticks, while schedule and droplet state remain the same. All actions that are deemed tickable by the Schedule in any given tick are indicated by an underscore.

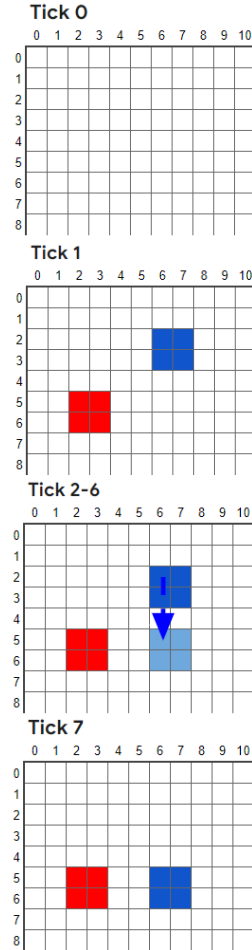
To determine which actions can be ticked, the Schedule checks all actions at the front of the queues, one action at a time. If an action has not yet been started, all droplets required by the action must be marked as AVAILABLE for the action to be able to be ticked. Note that the InputAction is the only action that does not require droplets to be AVAILABLE to be ticked.

If an action is already marked as IN_PROGRESS (i.e., started in a previous tick), the action is deemed tickable, regardless of the droplet status.

Whenever the Schedule is called, all actions marked as COMPLETED are popped from the queues. This is also the reason why actions are gradually disappearing from the schedule table in Figure 5.9 as actions are completed.

Each sub-figure shows the respective action-queues for each droplet. Actions in the queues are listed in the order they are to be executed, and each subsequent sub-figure illustrates the progression of the protocol through different ticks, showing the state of the grid and the status of the droplets at the beginning of each tick.

This example demonstrates how the scheduling algorithm ensures that actions are executed only when all required droplets are available, thus maintaining the high-level protocol's integrity. By following this example, one can see how the scheduling mechanism dynamically determine the actions that can be executed in any given tick based on droplet availability and action dependencies.



Initial Schedule. No droplets have been created yet.

	Droplet 1 (D1)	Droplet 2 (D2)	Droplet 3 (D3)
	NOT_CREATED	NOT_CREATED	NOT_CREATED
Queue(0)	Input(D1)	Input(D2)	Move(D3)
Queue(1)	Merge(D1, D2, D3)	Move(D2)	Output(D3)
Queue(2)		Merge(D1, D2, D3)	

After first tick, both input actions have been executed.

	Droplet 1 (D1)	Droplet 2 (D2)	Droplet 3 (D3)
	AVAILABLE	AVAILABLE	NOT_CREATED
Queue(0)	Merge(D1, D2, D3)	Move(D2)	Move(D3)
Queue(1)		Merge(D1, D2, D3)	Output(D3)
Queue(2)			

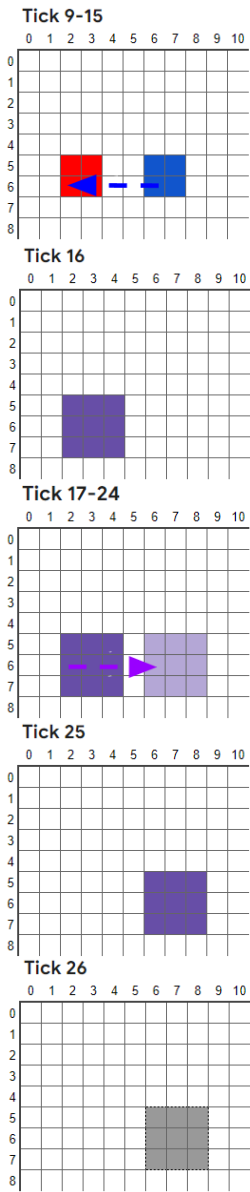
D2 has status UNAVAILABLE while moving.

	Droplet 1 (D1)	Droplet 2 (D2)	Droplet 3 (D3)
	AVAILABLE	UNAVAILABLE	NOT_CREATED
Queue(0)	Merge(D1, D2, D3)	Move(D2)	Move(D3)
Queue(1)		Merge(D1, D2, D3)	Output(D3)
Queue(2)			

D1 and D2 is both AVAILABLE after D2 is finished moving.

	Droplet 1 (D1)	Droplet 2 (D2)	Droplet 3 (D3)
	AVAILABLE	AVAILABLE	NOT_CREATED
Queue(0)	Merge(D1, D2, D3)	Merge(D1, D2, D3)	Move(D3)
Queue(1)			Output(D3)
Queue(2)			

Continues on next page



D1 and D2 has status UNAVAILABLE while merging.

	Droplet 1 (D1)	Droplet 2 (D2)	Droplet 3 (D3)
	UNAVAILABLE	UNAVAILABLE	NOT_CREATED
Queue(0)	Merge(D1, D2, D3)	Merge(D1, D2, D3)	Move(D3)
Queue(1)			Output(D3)
Queue(2)			

D3 has status AVAILABLE after succesfull merge.

	Droplet 1 (D1)	Droplet 2 (D2)	Droplet 3 (D3)
	CONSUMED	CONSUMED	AVAILABLE
Queue(0)			Move(D3)
Queue(1)			Output(D3)
Queue(2)			

D3 has status UNAVAILABLE while moving.

	Droplet 1 (D1)	Droplet 2 (D2)	Droplet 3 (D3)
	CONSUMED	CONSUMED	UNAVAILABLE
Queue(0)			Move(D3)
Queue(1)			Output(D3)
Queue(2)			

D3 is AVAILABLE when the move operation is COMPLETED.

	Droplet 1 (D1)	Droplet 2 (D2)	Droplet 3 (D3)
	CONSUMED	CONSUMED	AVAILABLE
Queue(0)			Output(D3)
Queue(1)			
Queue(2)			

D3 has been outputted, and is UNAVAILABLE.

	Droplet 1 (D1)	Droplet 2 (D2)	Droplet 3 (D3)
	CONSUMED	CONSUMED	UNAVAILABLE
Queue(0)			
Queue(1)			
Queue(2)			

Figure 5.9: Scheduling example

5.2.8 Compilation

The compilation process involves several important steps, to successfully transform a high-level protocol into BioAssembly. In this section, we will outline the compilation workflow, from parsing of data to handling of results.

Input Reading and initial setup

The compilation begins by reading the input data received from the frontend, which is covered in greater detail in Section 5.2.2. The data is consequently parsed into a `ProgramConfiguration` object. If no exceptions occur during the reading, an instance of `Executor` is created, provided with the aforementioned `ProgramConfiguration` object.

Executor Initialization

Upon initialization, the `Executor` uses the actions in the `ProgramConfiguration` to create a `Schedule`, as described in further details in Section 5.2.7. If no exceptions occur during this initial setup, the execution loop is initiated.

Execution Loop

During this phase, the `Executor` enters a `while(true)` loop, which continues until either of the following conditions are met:

1. **The Schedule is empty**, meaning that all actions has been marked as `COMPLETED`, indicating that the protocol has been successfully executed without any errors.
2. **An exception is encountered**, e.g. a deadlock occurred, which could not automatically be resolved by the system.

During each iteration of the `while` loop, the `Executor` attempts to execute a single tick by sequentially executing all actions deemed tickable by the `Schedule`. The results of each tick, is encapsulated in a `ActionTickResult`-object (see Figure 5.4 for class composition), and is hereinafter stored in a list of previously executed ticks `actionTickResults`. These results are saved in the order they were executed and can be used to generate the BioAssembly corresponding to the operations being performed in each tick.

If, during a tick, no actions is able to progress (i.e., the `ActionTickResult` object is empty), a deadlock has occurred. The `Executor` will then attempt to resolve the deadlock, a process which we will discuss further in Section 5.2.9.

Handling of results

Once the execution loop terminates, the `Executor` processes the results to generate the final output. Each `ActionTickResult` contains commands that describe the operations performed during the tick. These commands, as well as the interface '`IDmfCommand`', can be seen in Figure 5.4 from earlier.

The two primary commands implementing the `IDmfCommand` interface are the classes `SetElectrodeCommand` and `ClearElectrodeCommand`, which can generate the command to set and clear electrodes respectively. The `Executor` then combines these commands, to form the complete sequence of `BioAssembly` operations. An `ExecutionResult` object is created, which includes the aforementioned `BioAssembly`, as well as any relevant error messages encountered during execution.

Finally, the `ExecutionResult` object is returned to the REST API, which forwards the response to the frontend, which then facilitates the downloading of the relevant files, as well as presenting any error messages to the user, if needed.

5.2.9 Deadlocks

Deadlocks may occur, due to the constraint that droplets must not collide, unless explicitly intended (i.e. a `MergeAction`). For instance, if two droplets located besides each other on the platform needs to switch place, they may end up blocking each other, hence, a deadlock. In this section we will discuss how deadlocks are handled by the executor.

Deadlock Resolution

If a deadlock is detected (i.e., no actions are able to progress during a tick), the `Executor` will try to resolve it by randomly moving a droplet in a valid direction. This process is repeated up to a maximum of 1000 times. If a solution to the deadlock can not be determined within these 1000 attempts, an exception is thrown, explaining that an unsolvable deadlock situation has occurred. If a solution is found, the execution loop continues.

It is important to note that while the execution is usually deterministic, the random nature of how deadlocks is attempted resolved introduce some degree of non-determinism. This means that while simple protocols that can be executed without encountering any deadlocks may yield the same `BioAssembly` protocol for every execution, this is not necessarily the case for more complex protocols, where deadlocks are resolved during execution. More often than not, a deadlock can be resolved in multiple different ways. This means that a different solution may be found in different execution instances. Furthermore, since the number

of attempts is capped to 1000, it may be the case, although highly unlikely, that a execution attempt fails to find a solution while another correctly resolves the deadlock and completes the execution.

Deadlocks are notoriously complex and difficult to solve. By introducing a random element, we enable the executor to solve simple cases of deadlocks, though it may fail to resolve more involved deadlock situations. This is mitigated by the fact that a partial protocol, up to the point of the deadlock, is returned as `BioAssembly`, together with the error message. This allows the user to manually inspect the cause of the deadlock, e.g. by inserting it in the DMF Simulator mentioned in Section 2.1.4, and manually guide the executor by for example inserting `StoreActions` and/or rerouting droplets, to avoid the deadlock situation.

This solution equips the user with the necessary tools to manually resolve the deadlock by altering the protocol, while allowing the executor to try to automatically resolve simpler cases of deadlock situations with the help of random movements of droplets.

5.2.10 Exception Handling

Exception handling is an important aspect of ensuring the reliability of the compilation process. Throughout this thesis, we have discussed various instances where exceptions are thrown, and in this section, we will briefly outline how these exceptions are handled.

All exceptions explicitly being thrown in the backend (e.g. invalid protocols, deadlocks etc.) are all sub-classes of `DmfException`, which serves as the base exception class for all DMF-specific errors. The `DmfException` is itself a sub-class of the `RuntimeException`, allowing it to be thrown without being explicitly declared in method's 'throws' clause. The specific sub-classes include `DmfInputReaderException` and `DmfExecutorException`, to distinguish between errors being thrown during input processing and execution.

DmfExceptionMessage

The `DmfExceptionMessage` enum provides access to all relevant error-messages in a centralized manner, such that exception messages can easily be changed in a single class. `DmfExceptionMessage` facilitates the inclusion of context-specific variables in error messages, such as droplet IDs or specific positions, by overloading the `getMessage` method with various relevant parameters.

ExceptionHandler

The `ExceptionHandler` handles the logic of retrieving relevant error messages from an `Exception`. For instances of `DmfException` and consequently its subclasses, the `ExceptionHandler` simply retrieves the error message attached to the given exception. For all other exceptions, a generic error is returned, stating that a unknown error occurred during compilation.

It is important to note, that the program does not terminate immediately upon encountering an exception. Instead, the exception is caught, and the relevant exception message is attached to the resulting `ExecutionResult` object, which is returned to the frontend. Additionally, if an error is found during execution, the `BioAssembly` corresponding to the actions executed prior to the error, if any, is also included in the `ExecutionResult`. This, together with the appropriate error message, aids the user in debugging and consequently manually resolving any encountered errors.

5.2.11 Output Format

Upon termination, an `ExecutionResult` object is returned to the frontend through the established REST API. This object encapsulates the results of the execution process, including the compiled `BioAssembly` commands, the DMF configuration, and any relevant error messages encountered during the compilation.

The `ExecutionResult` object contains three relevant parameters:

1. **compiledProgram** - A string containing the `BioAssembly` protocol generated by the execution. Each `BioAssembly` instruction is separated by semicolons, as demanded by the `BioAssembly` ISA.
2. **dmfConfiguration** - A JSON object containing the structure of the provided DMF configuration used by the executor, with the addition of all droplets initially added to the platform through `Input` blocks. This allows the user to visualize the droplets on the simulator.
3. **errorMessage** - A string containing an error message to be displayed in the frontend. This field is empty if no errors were encountered during the compilation.

By ensuring a consistent structure regardless of the outcome of the compilation, the `ExecutionResult` can be easily recognized by the frontend, simplifying the handling of processing the received data greatly. The frontend simply adapts its way of handling the response according to the fields that contains data, and those which are empty.

CHAPTER 6

Testing and Evaluation

This chapter presents the methods used to test and evaluate the visual programming interface and execution and compilation engine, respectively. It describes the automated and manual testing procedures, as well as a real-world case study involving a dilution series to demonstrate the practical applications of the toolchain.

6.1 Automated Tests

Automated tests play an important role in ensuring the quality and reliability of software, especially in environments where multiple developers are working on a product simultaneously. Automated tests help catch bugs earlier, enforce code quality, and make the integration of new features more seamless.

When multiple developers contribute to a codebase, the chances of introducing errors increase. Automated tests provide a safety net that ensures changes and new features do not break existing functionality. Automated tests also allow for continuous integration, where changes and new features are regularly merged to the master branch, ensuring as few merge conflicts as possible.

We will now cover the implementation of automated tests for the frontend and backend individually.

6.1.1 Frontend

In the frontend, we have used Jest to create automated tests, most of which are in the form of unit tests. Jest allows for simple creation of automatic tests for JavaScript applications and of course also TypeScript as in this project, making it the obvious choice to use for this project.

In our frontend web-application, we have a test suite consisting of 163 automated tests. Overall, our tests in the frontend achieve a total line coverage of 89% with important business logic such as Redux reducers and validations nearly reaching or hitting 100% coverage. The high line coverage helps us to be confident in our code quality, especially when it comes to the more complex logic.

Unit Test Example: `removeBlock` Redux Reducer

An example of a unit test used to ensure correct behavior of the `removeBlock` reducer is presented in Listing 6.1. A `beforeEach` function runs before this unit test and provides an initial state of 3 code blocks. This unit test, amongst most other, is structured according to the three A's principle of unit testing: Arrange, Act and Assert.

- **Arrange:** First, the test is arranged by defining the index to remove and copying the initial state to a constant `prevState` to use for later comparison.
- **Act:** The test then acts by creating the constant `nextState`, which is the updated state.
- **Assert:** Lastly, the test asserts that the state has been updated as expected, by first checking that there are now 2 blocks in the array and that no index is selected (as the selected block has been removed). Then it compares the `prevState` and `nextState` to ensure that the indexes have been updated correctly and that the info of each block is still the same after the update.

For the many visual aspects of our web-app, automated testing cannot stand alone. Therefore, we have performed countless manual tests as described in greater detail in Section 6.2.1. However, even the visual aspects make sense to cover by automated tests. As an example, one of these tests is used to ensure correct rendering of the code blocks and arrows on the code canvas.

Listing 6.1: Unit test of removeBlock Redux reducer function

```

1 test("removeBlock", () => {
2   /* Arrange */
3   const indexToRemove: number = 0;
4   const prevState = { ...initialState };
5
6   /* Act */
7   const action = removeBlock(indexToRemove);
8   const nextState = blockReducer(initialState, action);
9
10  /* Assert */
11  // Check that array have been updated and no block is selected:
12  expect(nextState.blocks).toHaveLength(2);
13  expect(nextState.selectedIndex).toBe(null);
14
15  // Check that block indexes are updated correctly:
16  expect(nextState.blocks[0].index)
17    .toEqual(prevState.blocks[1].index - 1);
18  expect(nextState.blocks[1].index)
19    .toEqual(prevState.blocks[2].index - 1);
20
21  // Check that the new blocks contain the correct info:
22  expect(nextState.blocks[0].info).toEqual(prevState.blocks[1].info);
23  expect(nextState.blocks[1].info).toEqual(prevState.blocks[2].info);
24 });

```

6.1.2 Backend

For the backend, we used JUnit for automatic testing. JUnit is a widely-used testing framework for Java, that enables developers to annotate classes and methods as tests as well as offering powerful assertion methods, to verify that the outcome of a method is as expected.

There are currently 215 tests in the backend, achieving a total line coverage of 96%. Most of these tests are unit tests, which focus on testing the logic of each class individually.

To facilitate unit testing, we use Mockito, a popular mocking framework for Java. Mockito enables for creating mock objects with custom behaviour, allowing us to test each component in the project independently of each other.

Integration Tests

In addition to unit tests, the backend also houses a few select integration tests. These integration tests go beyond testing of individual classes, and is instead meant to test the interactions between multiple components. In these integration tests, a well-defined JSON object containing actions and a suiting

DMFConfiguration is parsed through the system. The resulting BioAssembly is analyzed, ensuring that the results are as expected. By not mocking throughout this process, everything from input reading, to BioAssembly generation is tested, promptly flagging any irregularities for the developer to look into.

Integration Test Example

To illustrate our integration testing process, we will in the following present an example of an integration test being performed in the backend, verifying that a single droplet can move 10 positions to the right. While this is a very simple test, it demonstrates the system's ability to produce a valid output, promptly flagging any issues in the basic functionality, if errors are introduced. More complex integration tests, such as testing the ability to compile a protocol where two droplets are input and moved concurrently, handling simple and unsolvable deadlocks, and a more involved case study (Dilution Series), are also performed.

The JSON protocol file used for this test, `SingleMoveRight.json`, mirrors the structure that would otherwise be received from the frontend. The content of the file is used to mock an API request from the frontend, ensuring the backend can correctly interpret and process the instructions, without relying on the frontend to send an actual API request. This file is saved locally in the testing environment within the source code and is hardcoded for the purpose of this test. The protocol can be seen in Listing 6.2.

Listing 6.2: `SingleMoveRight.json`

```
1 {  
2   "program_actions": [  
3     {  
4       "action": "input",  
5       "id": 0,  
6       "next": 1,  
7       "dropletId": "drop1",  
8       "posX": "10",  
9       "posY": "10",  
10      "volume": "7"  
11    },  
12    {  
13      "action": "move",  
14      "id": 1,  
15      "next": -1,  
16      "dropletId": "drop1",  
17      "posX": "20",  
18      "posY": "10"  
19    }  
20  ]  
21 }
```


The actions seen in Listing 6.2 defines an initial input action, placing a droplet at position (10, 10) with a volume of $7\mu\text{l}$, followed by a move action to position (20, 10). By not explicitly providing a 'dmf_configuration' section in the JSON structure, the executor will use the default configuration, matching the DMF platform presented in Section 2.1.1.

The corresponding integration test seen in Listing 6.3, verifies that this simple protocol is executed correctly:

Listing 6.3: SingleMoveRight.json

```

1  @Test
2  void integration_SingleDropletMove10Right() {
3      // Arrange
4      String jsonProtocolString;
5      try {
6          jsonProtocolString = /*load SingleMoveRight.json into string*/
7      } catch (IOException e) {
8          /* ... */
9      }
10
11     // Act
12     ExecutionResult executionResult =
13         ProgramExecutionService.executeProgram(jsonProtocolString);
14
15     // Assert
16     int dropletsInDmfConfig =
17         executionResult.getDmfConfiguration().get("droplets").size();
18     int setelCount =
19         executionResult.getCompiledProgram().split("SETELI").length -
20         1;
21     int clrelCount =
22         executionResult.getCompiledProgram().split("CLRELI").length -
23         1;
24     int ticks =
25         executionResult.getCompiledProgram().split("TICK").length - 1;
26
27     Assertions.assertNull(executionResult.getErrorMessage());
28     Assertions.assertEquals(1, dropletsInDmfConfig);
29     Assertions.assertEquals(24, setelCount);
30     Assertions.assertEquals(20, clrelCount);
31     Assertions.assertEquals(23, ticks);
32 }

```

Expected Results On the default DMF platform configuration, a droplet with a volume of $7\mu\text{l}$ is represented and moved by a 2×2 rectangle of electrodes on the grid. In the following, we will use this knowledge to determine the expected results seen in the 5 assertions clauses seen in Listing 6.3:

1. **Error Message** The execution should complete without any errors, meaning the error message should be null.

2. **Droplet Configuration** The resulting DMF Configuration should contain exactly one droplet, namely the droplet being inserted by the `Input` action.
3. **SETELI Commands** The initial placement of the droplet requires 4 electrodes to be set (2×2). To move the droplet 10 positions to the right, 20 electrodes must be set (`10 positions \times 2 electrodes wide`). Thus, the total number of 'SETELI' commands expected are $4 + 24 = 24$
4. **CLRELI Commands** Moving a 2×2 droplet 10 positions to the right, requires clearing a total of 20 electrodes (`10 positions \times 2 electrodes wide`).
5. **TICK Commands** The execution process involves 1 tick for the initial input action, 20 ticks for the movement (moving a droplet one position takes two ticks), and finally, 2 additional ticks need to be added after the concluding `TSTOP` command, to mark the ending of the protocol. Therefore, a total of $1 + 20 + 2 = 23$ ticks are expected.

While the integration test described above verifies the overall functionality of moving a droplet, it does not explicitly test which electrodes are set or cleared during the operation. This level of detail is covered by our unit tests, which test individual components such as electrode setting and clearing commands in isolation. By separating these concerns, we maintain a clear distinction between verifying the high-level interactions through integration tests and the detailed behaviour of individual components through unit tests. This approach ensures comprehensive coverage, allowing developers to promptly identify and address any issues in both the overall workflow and in the individual components of the compilation and execution engine.

6.2 Manual Tests

Manual testing, while often less systematic than automated testing, plays an important role in checking the usability and functionality of an application. This is particularly true for web-applications, where writing exhaustive automated tests to ensure proper layout and visual appearance can be very time-consuming, often without providing much value compared to manual testing of the visual appearance. Given the nature of the project being comprised of a frontend and a backend, we will cover the extent of manual tests in each of these, respectively, in the following.

6.2.1 Frontend

In the frontend we have performed numerous manual tests to ensure that the GUI both looks and behaves as expected. The main purpose of our manual tests have been to ensure proper layout and styling of components. While the business logic has been tested thoroughly through automated Jest tests, we have utilized manual tests to ensure that they do in fact behave the way in which they were intended to. A big part of the manual testing has consisted of running the web-app locally and perform small tests mid-development to ensure that the recent changes did not break the relevant areas of our web-app. Additionally end-to-end tests involving the backend has been performed, more on this in Section 6.2.3.

6.2.2 Backend

In the backend, manual tests were minimal. The primary manual testing involved was ensuring the functionality of the REST API by sending data, ensuring that the data was received, and that a valid response was returned. Additionally, we visually inspected prints of the electrode grid and path of the A-star algorithm in the terminal to ensure that these were as expected during development. This process could however be easily achieved, by performing an end-to-end test as described in Section 6.2.3.

6.2.3 End-to-End Tests

End-to-end test were performed numerous times throughout the project, involving every step in the process. These involve the creation of relevant protocols in the frontend, sending them to the backend for compilation, and inserting the generated files in the DMF Simulator, to visually inspect, that the protocol had been correctly converted into BioAssembly. End-to-end tests thus played a vital role in ensuring that the entire toolchain from frontend to backend to simulator works as intended.

6.3 Case Study: Dilution Series

A dilution series is a protocol commonly used in biochemistry to create a set of solutions with decreasing concentration of a particular substance. This process is vital in various applications, such as estimating the concentration (number of colonies, bacteria or viruses) in a specific sample of substance. This is achieved by simply counting the number of colonies cultured from serial dilutions of the sample and then backtracking the measured count to the concentration of the origin sample [28].

The dilution series protocol is illustrated in Figure 6.1, where each step represents a further dilution of the substance.

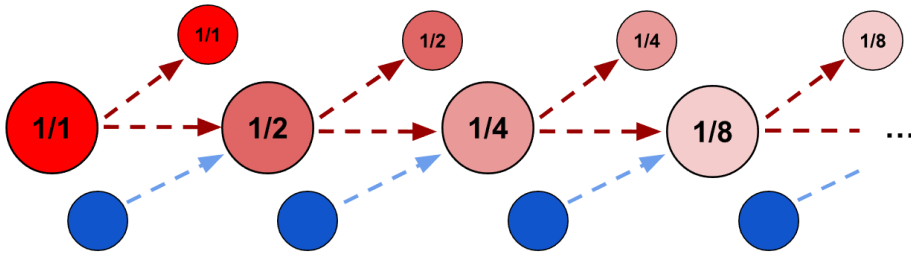


Figure 6.1: Illustration of a dilution series. The red circles represent the substance being diluted, and the blue circles represent the diluent. Each step shows the reduction in concentration.

In a dilution series, a substance is diluted multiple times in a stepwise manner, usually by mixing equal volumes of the substance and a diluent (typically water). The process involves repetitive merging, mixing and splitting of droplets, resulting in a series of solutions with a progressively lower concentrations of the diluted substance.

Stepwise Procedure

1. **Droplet insertions:** Initially, two droplets are inserted onto the platform. It is favorable, if the droplets are easily divisible using a series of halvings into a common volume.
2. **Droplets of equal size:** The origin droplets of diluent and substance are progressively split into smaller droplets, until they have the same volume.

3. **Merging droplets:** The two droplets of diluent and substance respectively are merged together, forming a new droplet twice the size, effectively halving the concentration of substance present in the resulting droplet, in respect to the origin substance droplet.
4. **Mixing droplet:** To ensure that the content of the previously merged droplets are mixed into a homogeneous substance, the combined droplet should be mixed well, i.e. by moving around in rectangles on the platform.
5. **Splitting of droplet:** The mixed droplet consisting of part diluent and part substance, is now split into two (identical) droplets, of the same size as the two origin droplets. The two resulting droplets, do the following:
 - (a) The first droplet is moved away, and outputted as a part of the dilution series.
 - (b) The second droplet repeats step 3-5 in a recursive fashion (taking the place of the substance droplet in step 3), until the desired number of dilutions has been performed.

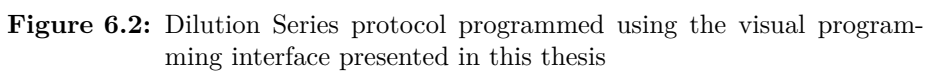
Mathematically, if C_0 is the initial concentration of the substance, after n dilutions, the concentration C_n can be expressed:

$$C_n = \frac{C_0}{2^n}$$

For example, after 5 dilutions, the concentration will be:

$$C_5 = \frac{C_0}{2^5} = \frac{C_0}{32}$$

The aforementioned protocol can be seen in Figure 6.2 where it has been implemented as blocks using the visual programming interface developed throughout this thesis. The first 13 blocks correspond to that of step (1) and (2). In these steps, the origin droplets of diluent and substance are inserted into the platform and divided into droplets of equal size, using a combination of **Input**, **Split**, and **Store** actions. In block 13 of the provided program, an initial droplet of 100% substance is being output, starting the dilution series. The steps (3), (4) and (5) can hereafter be clearly identified as a series of **Merge**, **Mix**, **Split**, and **Output** actions, resulting in a series of 9 droplets in total, ranging from 100% substance to $\frac{1}{2^8} = \frac{1}{256} \approx 0.39\%$ substance.



The dilution series protocol shown in Figure 6.2 consists of 45 blocks. When translated into BioAssembly, the protocol amass to a total of 3495 BioAssembly instructions, which is roughly 77 BioAssembly instructions per block of high-level actions. This significant increase in the number of instructions highlights the complexities and time consuming work that would otherwise be involved in manually programming such protocols directly in BioAssembly.

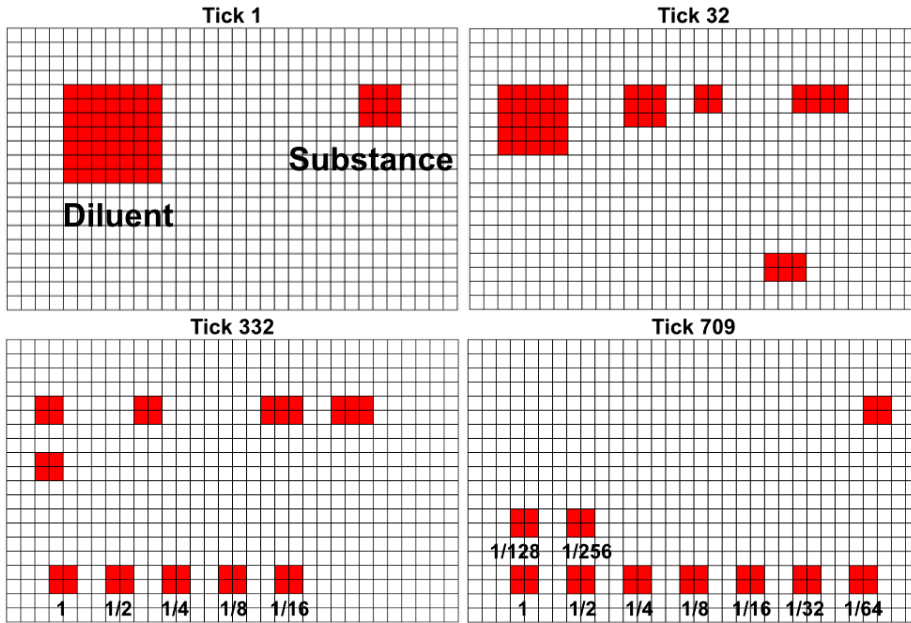


Figure 6.3: Screenshots from the simulator, simulating the dilution series protocol created using our toolchain.

The effectiveness of the dilution series protocol can be seen in Figure 6.3, which includes screenshots taken from the DMF Simulator. These screenshots demonstrate how the generated BioAssembly code accurately performs the dilution series, with the droplets progressively splitting and merging as expected.

The screenshots were taken at four different ticks of the simulation:

- **Tick 1:** Shows the state of the DMF platform after the initial Input actions. The big red droplet to the left indicate the diluent (i.e. water) and the smaller red droplet to the right, represents the droplet being diluted.

- **Tick 32:** Shows the initial splitting into droplets of equal size, as well as the initial merging of droplets.
- **Tick 332:** Displays an intermediate stage in the dilution process.
- **Tick 709:** Shows the final state where the resulting droplets in the bottom, are nine different droplets, with the first droplet being 100% substance and the last droplet being $\frac{1}{256}$ substance.

Additionally, a video demonstration of the dilution series being run on the DMF Simulator can be found at the following link:

Dilution Series Demonstration [YouTube Video]
<https://www.youtube.com/watch?v=QzGHZ8Laps8>

6.4 Discussion

The visual programming interface developed in this thesis offers significant advantages over manually programming BioAssembly protocols, as illustrated by the dilution series case study in Section 6.3. By automating the translation from high-level visual coding blocks to BioAssembly as performed by the compilation and execution engine, the speed and efficiency of which protocols can be created is increased significantly, especially for people with limited programming knowledge. By providing user feedback, both directly in the frontend, e.g. in the case of invalid droplet IDs, and in the backend, through exception handling during execution, along with the easier way of programming, our solution greatly reduces the potential for human error. This allows researchers to focus on experimental design and analysis rather than the intricacies of programming.

A potential issue with the programming interface in its current form is in regards to the scalability of the protocols. As protocols become larger and more complex, such as the dilution series protocol seen in Figure 6.2 consisting of 45 blocks, the visual representation of the protocol can become difficult to manage. While this issue could possibly be circumvented by future work looking into grouping of blocks along with the implementation of more instructions, such as a repeat instruction, it is important to remember that the 45 blocks are converted into an astonishing 3495 lines of BioAssembly.

Another discussion point is the visual versus text based programming languages. While visual programming languages offer an easier programming experience they can be constrained by the user interface, whereas text based programming

languages often offer a higher level of control, but at the cost of a steeper learning curve for people with no prior programming knowledge. In our project we focused on creating a programming interface for people with little to no knowledge of programming, which is why the visual way of programming was chosen.

The compilation and execution engine developed in this project provides great efficiency in both scheduling and finding optimal paths for the droplets. While it excels at what it does, it also has some room for improvement, especially in the area of handling deadlock situations. However, due to the scope and time frame for this project we limited the handling of deadlock situations to simple cases. Instead, the comprehensive error handling system aims to help the user easily debug the protocols themselves, either by reading the error message or simulating the protocol up to the point of failure, if possible.

Conclusion

7.1 Summary

In this thesis, we explored the design, development, and implementation of a visual programming interface for digital microfluidic biochips. The primary objective was to create a user-friendly toolchain that simplifies the programming of protocols to DMF biochips for users with limited coding experience. The visual programming language developed in this thesis, allows for the creation of high-level protocols, with the help of intuitive drag-and-drop blocks in a code-canvas, which are then translated into low-level BioAssembly code that can be run on the DMF platform.

The major contributions presented in this thesis can be summarized as follows:

- Creation of a visually appealing and intuitive graphical user interface (GUI) that enables users to create and manage complex DMF protocols through a visual block-based programming interface, increasing the usability of the DMF platform drastically for people with limited coding experience.
- Development of a robust backend system capable of compiling the high-level visual instructions created in the GUI into BioAssembly code that can

be interpreted by the DMF platform and DMF Simulator. The compilation and execution engine supports droplet scheduling, collision avoidance, attempted deadlock resolution, and error management.

- Support for various droplet operations: input, output, move, merge, split, mix, and store. All of which allows for creation of complex BioAssembly protocols that can be used to perform biochemical experiments at a small scale.
- Handling and intuitive display of errors in the frontend, both for errors that occur during execution of the protocol in the backend, but also for errors that occur directly in the frontend during creation of the protocol.
- Rigorous evaluation of the toolchain through a test suite consisting of 378 total automated tests between the frontend and backend along with numerous manual and end-to-end tests.
- A real-world case study of a dilution series, that demonstrates its effectiveness and practical utility. In the case study, 45 code blocks were used to generate 3495 lines of BioAssembly.

7.2 Future work

While this thesis has made substantial progress in the development of a visual programming tool for DMF platforms, there are several areas for future research and development:

- **Enhanced Scalability:** As the size and complexity of biochemical protocols increases, the presented visual programming interface may become difficult to manage. Future work could focus on improving scalability through features like block grouping and better organizational tools to manage large protocols.
- **Advanced Control Flow Blocks:** Introducing more sophisticated control flow operations such as loops, conditions, or functions would allow for more advanced and concise protocols, improving protocol design and readability.
- **Integration with Sensors and Actuators:** The current system primarily focuses on droplet manipulation. Integrating support for additional sensors and actuators could enable real-time feedback and adjustments, including temperature control, color sensors or other analytical tools.

- **Custom DMF Configuration Upload:** Currently, only a predefined DMF configuration matching the platform presented in this thesis can be used. Future work could enable the ability to upload custom DMF configurations directly from the frontend, allowing for greater flexibility and compatability with various DMF setups. This is currently possible in the backend by manually changing the file containing the default DMF configuration.
- **Deadlock Handling:** Further investigations could be made on the topic of resolving deadlocks. The current solution relies on random attempts, and is only able to solve primitive deadlocks using this method. Future research should focus on developing a more structured and reliable approach, allowing automatic resolution of more complex deadlock scenarios.

By addressing these areas, future developments can build upon the foundation laid by the solution presented in this thesis, further increasing the capabilities and usability of programming for digital microfluidic biochips.

Bibliography

- [1] Paul Pop, Mirela Alistar, Elena Stuart, and Jan Madsen. *Fault-Tolerant Digital Microfluidic Biochips*. Springer, 2016.
- [2] Paul Pop, Wajid Hassan Minhass, and Jan Madsen. *Microfluidic Very Large Scale Integration (VLSI)*. Springer, 2016.
- [3] Bioassembly: A parallel synchronous instruction set architecture for digital microfluidics.
- [4] Georgi Tanev, Luca Pezzarossa, Winnie E. Svendsen, and Jan Madsen. A Reconfigurable Digital Microfluidics Platform. 2019.
- [5] Blockly. <https://developers.google.com/blockly>. Accessed: 02-06-2024.
- [6] Image of lego mindstorms programming interface (thumbnail of youtube video). <https://www.youtube.com/watch?app=desktop&v=9HiNF1ry6x0>. Accessed: 26-05-2024.
- [7] Typescript homepage. <https://www.typescriptlang.org/>. Accessed: 06-05-2024.
- [8] React homepage. <https://react.dev/>. Accessed: 06-05-2024.
- [9] npm homepage. <https://docs.npmjs.com/about-npm>. Accessed: 06-05-2024.
- [10] Redux homepage. <https://redux.js.org/>. Accessed: 06-05-2024.
- [11] Redux toolkit. <https://redux.js.org/tutorials/quick-start>. Accessed: 20-05-2024.

- [12] Axios. <https://axios-http.com/docs/intro>. Accessed: 26-05-2024.
- [13] React-dnd homepage. <https://react-dnd.github.io/react-dnd/about>. Accessed: 06-05-2024.
- [14] Jest. <https://jestjs.io/>. Accessed: 06-05-2024.
- [15] Mui components. <https://mui.com/material-ui/all-components/>. Accessed: 06-05-2024.
- [16] Mui icons. <https://mui.com/material-ui/material-icons/>. Accessed: 06-05-2024.
- [17] Tailwind css homepage. <https://tailwindcss.com/>. Accessed: 06-05-2024.
- [18] Oracle, java homepage. <https://www.oracle.com/java/>. Accessed: 06-05-2024.
- [19] Spring boot. <https://spring.io/projects/spring-boot>. Accessed: 06-05-2024.
- [20] Information about rest and restful apis. <https://aws.amazon.com/what-is/restful-api/>. Accessed: 02-06-2024.
- [21] Jackson. <https://github.com/FasterXML/jackson>. Accessed: 06-05-2024.
- [22] Lombok. <https://projectlombok.org/>. Accessed: 06-05-2024.
- [23] Junit. <https://junit.org/junit4/>. Accessed: 06-05-2024.
- [24] Github. <https://github.com/>. Accessed: 06-05-2024.
- [25] Figma. <https://www.figma.com/>. Accessed: 10-05-2024.
- [26] A-star search algorithm. <https://www.geeksforgeeks.org/a-search-algorithm/>. Accessed: 26-05-2024.
- [27] Admissibility of a-star algorithm. <https://www.geeksforgeeks.org/a-is-admissible/>. Accessed: 22-05-2024.
- [28] Avishai Ben-David and Charles E. Davidson. Estimation method for serial dilution experiments. 2014.