DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Front-End Web Development of an Interactive Design Platform for Microfluidic IP Modules

Xianer Chen

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Front-End Web Development of an Interactive Design Platform for Microfluidic IP Modules

Front-End-Webentwicklung einer interaktiven Designplattform für mikrofluidische IP-Module

Author:	Xianer Chen
Supervisor:	Prof. Dr.-Ing. Ulf Schlichtmann
Advisor:	M.Sc Yushen Zhang
Submission Date:	15. July 2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15. July 2022 Xianer Chen

Acknowledgments

Many thanks go to Chair of Electronic Design Automation (EDA) of the Technical University of Munich (TUM) for hosting this thesis, especially to Prof. Dr.-Ing. Ulf Schlichtmann as the supervisor. Special thanks go to team Columba from TUM, whose research inspired this thesis.

Most of all, I want to thank M.Sc Yushen Zhang for offering me the opportunity to work on an inspiring practical front-end project and for guiding me throughout the work. Without his support this thesis would not have been possible, and I cannot be more grateful for it.

I would also like to thank my family and friends for their support throughout my undergraduate studies, especially during the difficult times of the pandemic.

Abstract

With the rise of microfluidic chips in the field of biotechnology over the past decade, the functionality and structure of the microfluidic chips are gaining increasing complexity, which complicates the process of designing them. In recent years, researchers have aimed to replace the manual design process with an automated design process for complicated microfluidic devices and proposed promising solutions. However, the applicable scope of such design automation software is highly dependent on the variety of microfluidic modules it can provide. As new types of microfluidic components have been continuously developed nowadays, the module library of the design automation software needs to be constantly updated. With conventional design software like AutoCAD, module developers need to manually describe layer interactions within the module, making the design process very inefficient.

This thesis will introduce Module Sketch, an interactive web-based design platform for microfluidic components. Module Sketch provides a comprehensive solution to simplify the work of designing new components and an easily accessible platform to support the development of microfluidic intellectual property (IP) modules.

Contents

1 Introduction

Continuous-flow microfluidic large-scale integration (mLSI) [1] has been a rapidly growing field over the past decade and is widely used in biomedicine and biochemistry. Composed of various microfluidic components and interconnected microchannels, microfluidic biochips integrate the laboratory experiments of chemical synthesis and biological analysis [2], including sampling, dilution, reaction, separation, etc. Thus, it is also known as Lab-on-a-chip (LoC) [3]. Through precise manipulation of small fluids, especially on the sub-millimeter scale [2], LoC can greatly shorten the sample processing time and achieve the maximum utilization efficiency of reaction samples and reagents, which provides extremely broad prospects in biological and medical fields.

As the potential of mLSI biochips is being exploited, they are gaining increasing complexity in structure and features. However, similar to the early development of integrated circuits, the disadvantages of manual design—time-consuming and error-prone—became more obvious, which is inevitable for the manual design of highly complex mLSI chips. In recent years, researchers of design automation for mLSI have been making attempts to replace the conventional AutoCAD manual design process [4], and now a promising solution has emerged.

Cloud Columba[1], developed by Tseng and colleagues from the Technical University of Munich (TUM), can interpret users' input of device specifications and generate optimized designs of the chip layouts automatically [5]. As shown in Figure 1.1, Columba provides a user interface where users can specify the type, quantity, and some specific properties of the required components. These components are defined in Columba's module library, *module models* [6, 4]. For now, Columba has two different modules available for the user: ring mixer and chamber container, but these can only provide the most basic features of mLSI. Columba's researchers believe that it is necessary to expand the module library by introducing more

[1] https://cloud-columba.org/

module model categories [7]. However, they are facing the same problem as described earlier in the mLSI chip design. At the moment, in order to design a new component, designers have to draw the design first on AutoCAD and then interpret it into a .JSON file that conforms to the format of the module models. On the one hand, components are very likely to have two layers interacting with each other, and designers cannot define the layer interaction on AutoCAD. On the other hand, the design drawings cannot be processed into the required format automatically. Manual interpretation is tedious and will only add to the time and personal costs. Thus, it goes against the initial intention of design automation.

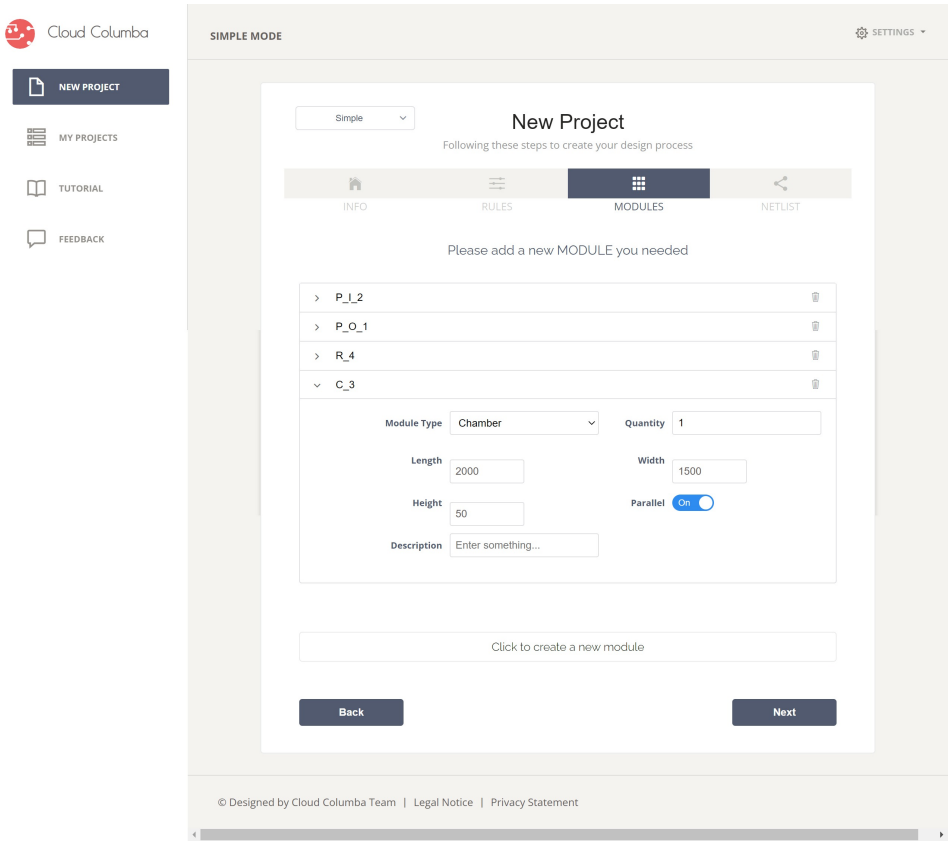


Figure 1.1: UI of Cloud Columba.

Nonetheless, Columba's concept still holds great potential, as it is able to perform the optimization of the mLSI layout on its own without human intervention. But in order to expand its module library, Columba's researchers need a dedicated module design tool that can solve the problems discussed above.

In this project, we propose Module Sketch, a web-based interactive design platform for

mLSI components. It is convenient to operate and provides comprehensive drafting features similar to AutoCAD-like applications. As a highly targeted module design application for Columba, it allows users to describe how the component could be integrated into microfluidic chips, and generate a design output as a .JSON file which is guaranteed to meet the rules and standards of Columba's module models.

2 Background

In this chapter, we will take a closer look at the fluid control mechanism of mLSI, and then we will provide a general introduction to front-end technology. At last, we will have a review of the computer-aided design (CAD) for mLSI [5] currently in use.

2.1 Fluid Control of Continuous Microfluidics

As mentioned before, the most remarkable feature of mLSI is its precise fluid control, which is supported by the interaction of microfluidic layers. Although the fluid control mechanism in single-layered microfluidic chips has recently begun to be explored [8], interactions between the microfluidic flow layer and control layer are still the mainstream way to fulfill this feature. The reaction samples and reagents are located in the flow layer, flowing along the channels within [9]. Made from polydimethylsiloxane (PDMS), microchannels have an elastic mechanical property and can therefore deform when pressure is applied. The control layer lies above the flow layer and consists of control channels and pneumatic valves, which are placed on top of the flow channels. When activated by air pressure, the valve squeezes the flow channel and blocks the fluid within. Correspondingly, when the air pressure is released, the blockage will be lifted, which will allow the fluid to be transported further [9].

As part of the continuous microfluidic chip, a microfluidic module also contains the aforementioned layer-to-layer interactions. Figure 2.1 [4] shows an example of a continuous microfluidic chip design, where the flow channels are denoted by red color, and the control channels and the valves are denoted by green color. (a) and (b) are two different components that involve the interaction between layers.

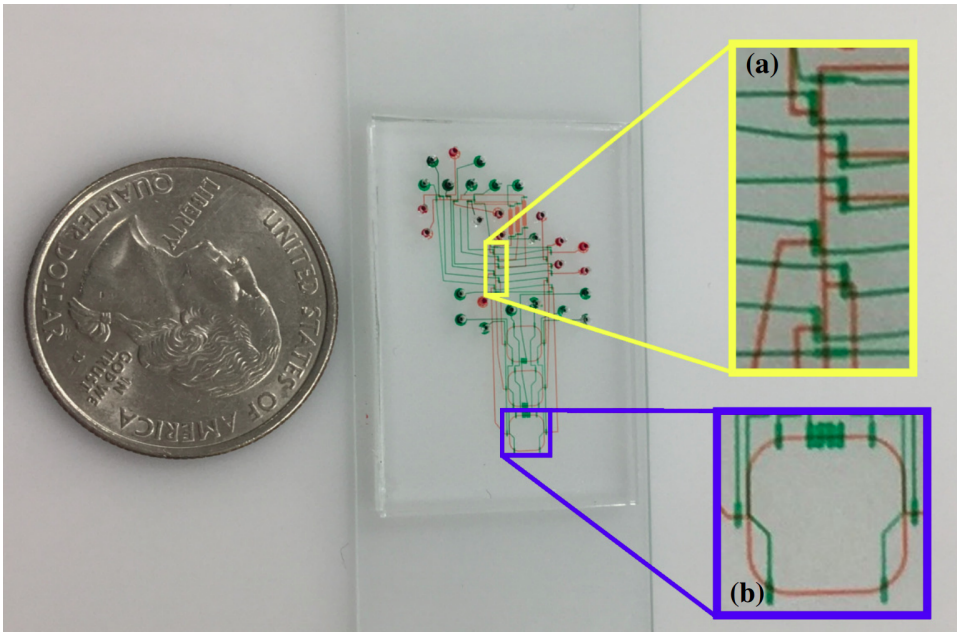


Figure 2.1: A design generated by Columba 2.0 [4]. (a) A switch. (b) A rotary mixer.

2.2 Front-End Techniques

The development of the front-end is a reflection of the development of the Internet itself. Since the inception of the Internet, front-end developers have been working on enriching the content and the interactivity of web pages in pursuit of the best user experience.

Hyper Text Markup Language (HTML) [10], Cascading Style Sheets (CSS), and JavaScript have been the core of front-end development since the very early days of the Internet. At present, HTML6 is on the horizon, but we can still recall the revolution that HTML5 started in October 2014 through a variety of new features, including: the multimedia elements `<audio>` and `<video>` and the graphic elements `<svg>` and `<canvas>`. JavaScript was standardized by European Computer Manufacturers Association (ECMA) since 1997 under the alias ECMAScript [11]. It can dynamically modify the Document Object Model (DOM) structure in response to user events on the browser. In the mid-2000s, the Internet entered the era of Web 2.0, where the concept of Asynchronous JavaScript and XML (AJAX) [12] played an important role. AJAX made it possible to update the web content without reloading the whole page and is widely used by popular applications, such as Google Maps and Gmail.

For a large-scale front-end project, complex interactions and massive logic processing can

result in a huge amount of JavaScript code, which increases the difficulty of development and maintenance. Hence the Model View Control (MVC) framework is proposed to simplify the front-end development. Through reasonable organization and layering, the front-end code becomes cleaner and logically clearer, making the functionality enhancements and testing much more convenient. Currently, React.js, Angular.js, and Vue.js are the three most commonly used front-end MVC frameworks.

2.2.1 Scalable Vector Graphics

Among the features of HTML, Scalable Vector Graphics (SVG) plays the most important part in our application.

SVG is an Extensible Markup Language (XML). It has been developed by the World Wide Web Consortium (W3C) since 1999 [13]. An SVG document, which consists of only the <svg> root element and shape elements inside of it, can be inserted almost anywhere in the HTML body. The user can not only easily scale, rotate or translate SVG but also apply filters on it or use SVG for animation.

As the name suggests, SVG is used to represent vector graphics. Unlike bitmaps, which are a bunch of pixels, vector graphics are independent of resolution and, therefore, can be rendered at any size without any loss of quality. This feature is critical for our design tool. In order to draft a precise design, users need to constantly zoom in and out on the parts they are working on. Design in the work area should always be clearly presented to the users and SVG has an absolute advantage in this regard.

Another benefit of the <svg> element is that it has a DOM interface, and therefore we can attach JavaScript event handlers to <svg> elements or the elements inside the <svg><\svg> tags, which makes them more convenient to modify.

2.2.2 JavaScript

JavaScript is a scripting language that is also one of the three core technologies of the World Wide Web (WWW). It was invented by Brendan Eich in 1995 [14] to add interactivities to websites. JavaScript allows developers to capture users' activities on the browser, such as

mouse, keyboard, and wheel activities, and bind customized event handlers to them. It provides local storage to store data and methods to modify the content or the style of the website. Since 2009, with the release of Node.js [15], JavaScript has transformed from a language that ran purely on the client-side to one that is equally useful on the server.

According to a recent survey published by SlashData [16], JavaScript remains the most popular programming language for the tenth survey in a row, with close to 17.5M developers worldwide using it. GitHub's 2021 Octoverse Report [17] showed similar results (Figure 2.2).

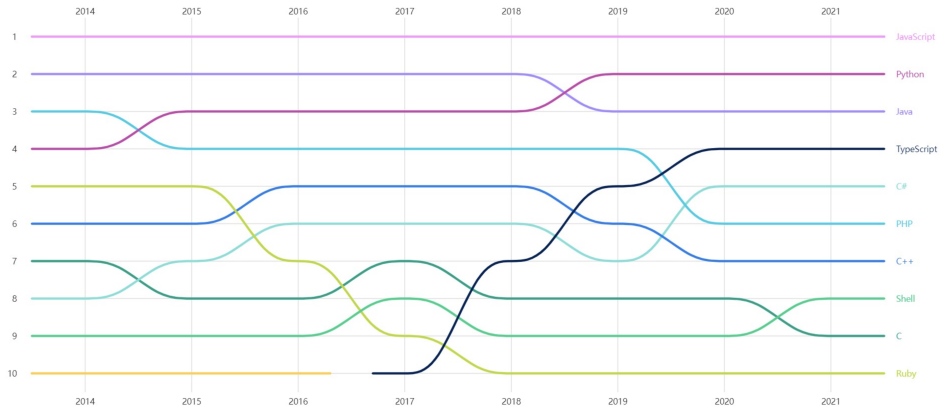


Figure 2.2: "Top languages over the years" from Octoverse Report 2021 [17].

2.2.3 Front-End Frameworks

After JavaScript laid a path for the dynamic interaction of web pages, in order to establish the user interface of the websites quickly and simplify the development, front-end frameworks emerged, which are usually based on JavaScript or TypeScript. According to the survey result of State of JS 2021 (Figure 2.3) [18], the mainstream frameworks and libraries in the front-end field are still React, Vue and Angular, while some new frameworks like Svelte also have increasing competitiveness.

React was created and maintained by Meta (formerly Facebook) [19]. It was first used for establishing Facebook's internal products like Instagram and then released to the public in 2013. It supports component reuse and is attracting the attention of more and more developers.

Angular [20] is the successor of AngularJS (both owned by Google). With Angular, devel-

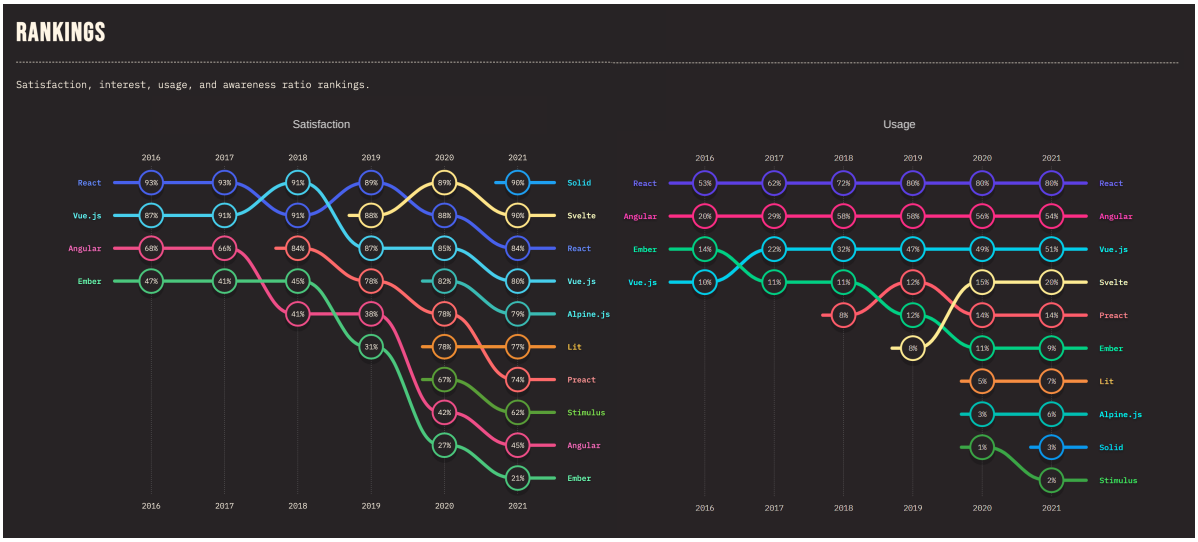


Figure 2.3: Rankings of front-end frameworks from State of JS 2021 [18].

opers can easily build complex websites with user-defined components. Angular is equipped with a large ecosystem, and it supports two-way data binding, which means that if the value of a variable in the model changes, the DOM tree (view) will be modified accordingly, and vice versa.

Vue's creator Evan You used to work at Google using AngularJS [21]. Hence Vue inherited the advantages of the powerful AngularJS, including the component-based pattern and two-way data binding [22]. Another benefit of Vue is that it is more simplified and lightweight, which is the reason why we use Vue as the framework in this project. Meanwhile, Vue is also perfectly capable of supporting complex single-page application (SPA) development when used with related tools and libraries, such as Vuex, a state management[1] pattern for Vue applications.

[1]Pinia has recently replaced Vuex as the official state management library for Vue. For more information see:
https://vuex.vuejs.org/
https://pinia.vuejs.org/introduction.html

2.3 Design Tools of mLSI

2.3.1 Cloud Columba

In the last chapter, we briefly discussed Cloud Columba, the design automation software for continuous microfluidics developed by Tseng's team from TUM. For now, three versions have been successively released: Columba, Columba 2.0, and Columba S [6, 4, 23]. Each version has a significant improvement over the previous one, which shows great potential in mLSI manufacture.

As mLSI is the integration of microfluidic modules, the usability of design automation is highly dependent on the variety of its module library, which is the module models in the scope of Columba.

Each module model has well-defined flow and control channels within a rectangle-shaped boundary. All the junctions, where channels can be connected to external devices, lie on the boundary [4]. What Columba does is decide how the module is geometrically integrated onto the chip, and which junctions are used as connections to other devices.

Figure 2.4 (a) shows the schematic of the module model for a ring mixer. Flow channels are denoted by blue color. The left and right sides are connected to the external of the module. Control channels are denoted by light green color and valves by orange, pink, or dark green color. Each valve could be bound to one or several control pins and have different binding options. For example, (b)(c)(d) are the three binding options for the same valve, which is bound to the sixth and seventh control pin. Binding options will be further discussed in section 3.2.

2.3.2 3DµF

3DμF[2] is a web-based open-source interactive microfluidic system designer developed by CIDAR lab[3]. On 3DμF, users can modify continuous microfluidic designs easily with drag-and-drop. 3DμF provides a library of useful microfluidic components with modifiable properties, as well as the option to import user-defined components from external .DXF files

[2] https://3duf.org/
[3] https://www.cidarlab.org/

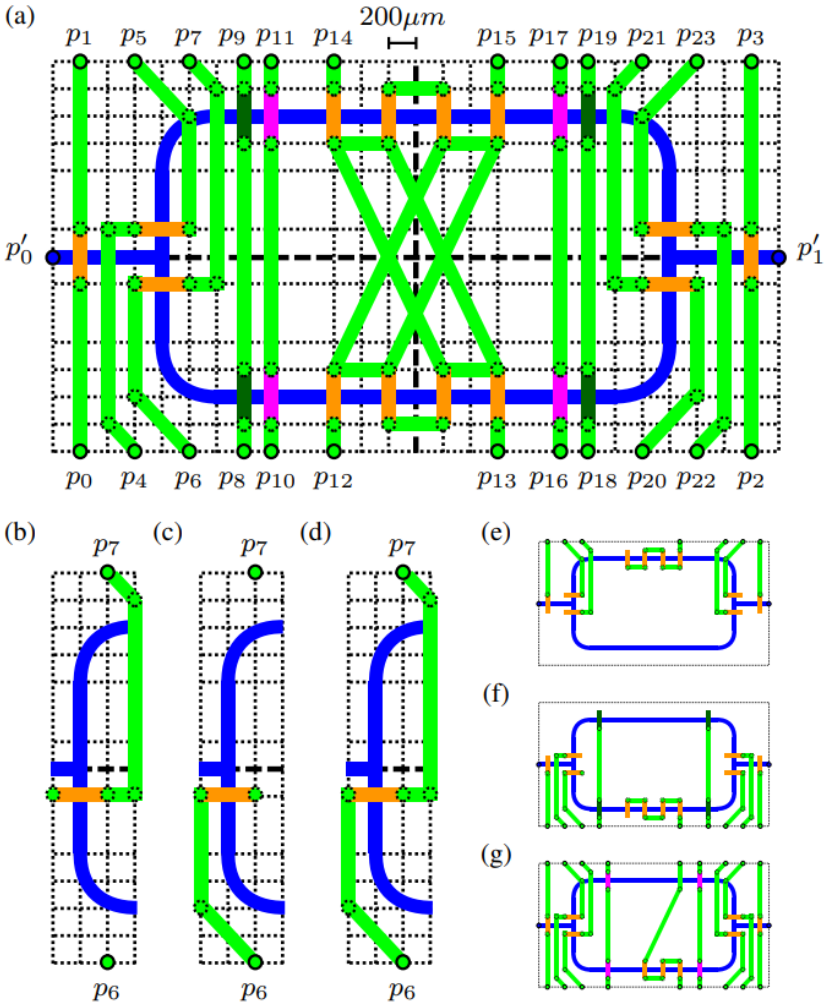


Figure 2.4: Module model for a ring container.

[24].

2.3.3 AutoCAD

AutoCAD is a fully equipped design tool widely applied in various industries. From machinery manufacturing to architectural design, AutoCAD is the universal solution for design drafting. The same goes for continuous microfluidics chips, and attempts have been made to make AutoCAD more applicable. For instance, in 2009, Amin's team from the Massachusetts Institute of Technology (MIT) proposed an AutoCAD-plugin named Micado [25], which supports automatic valve placement and routing for the control layer of simple continuous microfluidic chips.

However, our goal is to develop a module design platform for Columba. Although with AutoCAD alone, users can perfectly present the structure of the components to be designed, they cannot describe control bindings or generate output in the format that Columba desires. On the other hand, compared to web applications, AutoCAD requires an installation, which makes it less convenient to access. Despite these flaws, AutoCAD still provides comprehensive drafting functionalities and a user-friendly interface, which is worth for reference.

3 Functional Requirements Analysis

In this chapter, we will analyze the functional requirements of this project, from the basic features of general drafting software to the ones that are particularly required for microfluidic components design. First of all, we would like to briefly introduce the workflow of our application through a simple activity diagram (Figure 3.1). In subsections, we will be discussing each activity in detail.

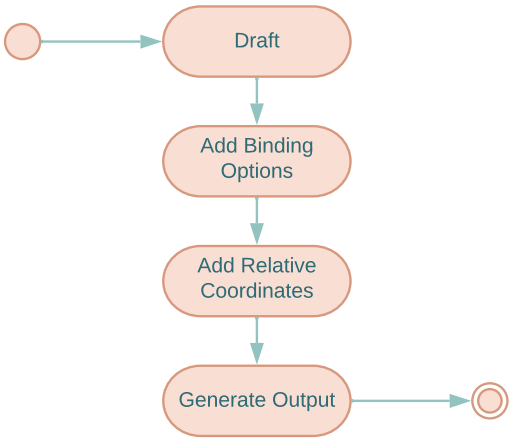


Figure 3.1: A simple activity diagram of Module Sketch

Note that the sequence of activities in the diagram is desired but not strictly specified in practice, as the user may need to adjust the previous steps at any time. However, it should be clarified that each activity in the diagram has a certain degree of dependence on the previous ones. For example, in the activity *Add Binding Options*, the user needs to select figures that have been drafted. In the meantime, the user can still step back and delete a shape that has been added to a binding option, but our application will then modify the binding options accordingly, since it cannot allow an undefined figure to remain in the system.

3.1 Drafting

As discussed before, a continuous microfluidic component usually consists of two layers: a flow layer and a control layer. For both layers, drafting is an essential functionality, where users draw up the designs from scratch by adding and editing entities that represent chambers, channels, and valves.

In order to optimize our user experience in drafting, we need to clarify a lot of details, although many of them may seem intuitive to users nowadays. As we are building our App from scratch, every detail that may influence user experience matters and is worth our extra thoughts and attention. The central idea is that a good application should act the same way as the user would expect.

Shape Drawing: We simplify the entity design process to shape drawing since the component parts like chambers and channels can be represented by shapes like solid rectangles and polylines, respectively. Users should be able to draw basic solid and hollow figures, such as rectangles, rounded rectangles, ellipses, polygons, and polylines, and by combining these basic shapes, they can create complex figures.

Shape Selecting & Editing: The shapes in the work area should be flexible. The selected shapes will be highlighted, and have assisting points around them, which users can drag and drop for editing operations, such as moving and resizing.

Group Selecting: If multiple shapes are supposed to be edited in the same way, like moved by the same offsets or copied, we should not ask the user to repeat the same operation for each shape. The user may need to select a group of shapes and move or copy them.

Switching between layers: As stated before, if a component has two layers, the user may need to design them separately. When the user is operating on one layer, the other layer should be automatically set as inactive, meaning that it cannot be accessed, but it is still visible with less opacity so that the user can have an overall observation.

Zooming and Panning: Zooming and panning can either be achieved with the mouse wheel (like AutoCAD) or by clicking buttons or with hotkeys (like Adobe Photoshop or Illustrator). Experienced users would prefer the mouse wheel or hotkeys, but for inexperienced users, buttons are the most straightforward solution. Both are worth

taking into account.

Pins: Users may need to mark some points with notations, such as inlets or outlets of the fluids. Pins are only visible during the designing process and will be ignored in output generation.

Copy and Paste: For the shapes that may appear multiple times or in a pattern, copy and paste will save the repetitive work.

Redo and Undo: It is necessary for a design tool to enable users to discard and resume previous operations.

Snap: The vertices of shapes or midpoints of line segments are often critical points that users may want to reuse. Making them snap points will help the user to find the exact position for drawing and editing.

Ctrl and Shift key: Ctrl and Shift keys are assigned various functions and are widely used on all kinds of user interfaces. Besides hotkeys like *Copy* (`Ctrl+C`) and *Paste* (`Ctrl+V`), a combination of Ctrl or Shift and mouse click can also modify the selection by adding or removing the item from the selected group. In addition, on design tools like Adobe Illustrator, pressing the Shift key while drawing may add constraints to the shape and force users to draw only horizontal or vertical lines, perfect circles, squares, etc.

3.2 Control Bindings

So far, we have covered almost all the features of the flow layer, but in the control layer, users will additionally describe the behavior of valves and control channels.

Cloud Columba treats every component as a rectangular block with control pins on the boundary. Every control channel is connected to its unique control pin and the valves that it controls. In most cases, a valve can be controlled by multiple pins. For this concept, we use the term *binding options* for the valve, and we call the task of choosing one viable binding option for a valve or valve group (some valves only exist in groups) a *control task*. An important part of the work of Cloud Columba is to complete each control task for all the components on the mLSI. In short, we must design a way to describe the control tasks for the component to

make sure that Columba knows everything needed for this process.

It is straightforward to add binding options for a **single independent** valve by following the steps shown below:

1. Select a valve.
2. Start adding a new binding option.
3. Select channels.
4. Finish adding the binding option (or discard selection).

We specify **single** and **independent** because the complexity of adding binding options will greatly increase in other cases. Currently, there are two typical situations:

- **Pumps**
  As mentioned above, some valves do not exist alone but are only concatenated together into a group by control channels. Such a valve group is usually called a *pump*. Users can form a pump by selecting the valves from the work area, and then the pump can be treated like a single independent valve, meaning that users can add binding options for it through the process given above.

- **Linked valves**
  In contrast to independent valves, if two valves are linked together, their binding options will be treated as one control task, meaning that only one of them can exist in the final design. This procedure works differently from creating a group: when the user forms a group, the chosen valves can still be added to other groups. However, a valve or a valve group can only be linked once.

Note that both situations above may occur simultaneously, that is, users may need to link valve groups. Take Columba's mixer as an example (Figure 2.4), (e), (f) and (g) are three possible implementations of the mixer in the final layout. We can see in (f) and (g) that both implementations have the pump in the middle of the bottom of the mixer. The pump supports two different physical structures to enable flexible pin access. Namely, it is composed of either three or four valves. In this case, the user needs to form two groups, namely one group of three valves and one of four, to represent this pump. This raises a problem since the two groups are not independent of each other because we cannot choose one binding option for

each of them. In other words, the two groups must be linked together so that they belong to the same control task.

3.3 Relative Coordinate & Output Generation

On the microfluidic chip design software, components are mostly variable in size, and some channels can be extended to make the component scalable. Thus, in our application, the figures displayed in the work area do not necessarily match the corresponding shapes in the final microfluidic chip design. To support the extensible channels, we have to introduce the concept of *relative coordinate* (denoted by `xRel`, `yRel`) and, accordingly, *absolute coordinate* (`xAbs`, `yAbs`).

In the representation of a component, every point has its unique absolute and relative coordinates. The relative coordinate indicates the position of a reference point with respect to the whole design layout. Both `xRel` and `yRel` have the value of a rational number between 0 and 1. Take `xRel` as an example, the value 0 represents the leftmost end of the design, and value 1 is the rightmost end. The absolute coordinate represents the distance between the point and its reference point.

Given the position—the top left corner—of a component on an LoC (X, Y), the size of the component ($width \times height$) and a point P on the component with relative coordinate (`xRel`, `yRel`) and absolute coordinate (`xAbs`, `yAbs`), then the coordinate of P on the LoC ($P.x$, $P.y$) can be calculated as follows:

$$\begin{pmatrix} P.x \\ P.y \end{pmatrix} := \begin{pmatrix} X + xRel * width + xAbs \\ Y + yRel * height + yAbs \end{pmatrix} \tag{3.1}$$

After the user defines relative coordinates for every point in the component design, the entire design process is complete. Then a .JSON file can be exported, which contains all the information of the design. Listing 3.1 shows the structure of the .JSON output.

Listing 3.1: The structure of the output

```
node := {"xAbs": double, "yAbs": double, "xRel": double, "yRel": double}

valve := {...node, "ori": string-orientation}
```

```
arc := {"type": "arc", ...node, "angle1": double, "angle2": double}
line := {"type": "line", "point": [...node...]}
area := {
    "type": "solid",
    "start": node,
    "end": node,
    "path": [...(arc|line)...]
}

controlOption := {
    "control_pin_id": [...int...],
    "channel": [...line...],
    "valve": [...valve...]
}

controlTask := {"option": [...controlOption...], "id": int}

module := {
    "name": string-title,
    "control_element": [...controlTask...],
    "flow_layer": {"shape": [...(arc|line|area)...]}
}
```

4 Design & Implementation

In this chapter, we will discuss the specific design and implementation of Module Sketch and how it works behind the scenes.

4.1 User Interface

To start with this chapter, first, we would like to introduce our user interface (Figure 4.1).

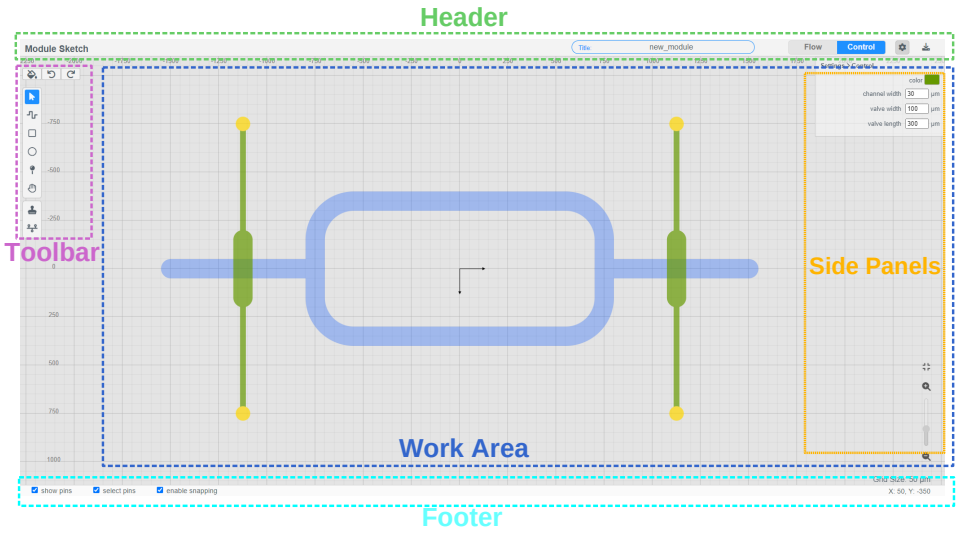


Figure 4.1: The user interface of Module Sketch.

Header - The header consists of (from left to right) a text field to set the title and three buttons, respectively: one for switching between layers, one for toggling layer settings, and another to start output generation.

Footer - The footer has some checkboxes for settings on the left side, and presents (rounded) cursor coordinate on the right side.

Toolbars - The main toolbar shows the modes of our application, which are (in top-down order) select, polyline/polygon, rectangle, ellipse, pin, pan, and two additional modes only available for the control layer: valve and bind. The top toolbar has three buttons. With the first one, user can toggle between area mode and channel mode, which can affect the selected shapes or new shapes to be drawn. Redo/undo can be triggered by the next two buttons, as well as by hotkey `Ctrl+Y` and `Ctrl+Z`.

Work Area - The work area occupies almost the entire page and is where all SVG elements are rendered. Thus, most event listeners are assigned to the work area. It consists of multiple layers of SVG, including SVG of snap points and assisting points for editing. Snap points and assisting points must be rendered above the content. Otherwise, they might be blocked by other shapes and cannot catch any user events. On the bottom is the background grid, which assists the user in aligning.

Side Panels - The panels on the right side imply the current status. On the layer settings panel, users can adjust color, channel width, and the size of valves. Panels for modification of control bindings and relative coordinates will also be mounted here. In the lower right corner, there is an area for the presentation of the zooming status with the current grid size and buttons for zooming-in, -out, and resetting the view.

4.2 Code Structure

The code of Module Sketch follows the default Vue3 project structure and the official style guide[1], namely, we keep our project directory flat and avoid reduplicative nesting (Figure 4.2).

As we are building a SPA, `index.html` is the start site and also the only .html file in the directory. Its body contains one single element `<div id="app"></div>`, where the root component `App.vue` will be mounted. Figure 4.3 shows the Vue component tree of `App.vue`.

For a large-scaled SPA with nested components that share the same states, we need to pass the states from the parent components to the children as properties, and when a child wants

[1] https://v2.vuejs.org/v2/style-guide/

```
module-sketch
│
├─public            // index.html
└─src
    │  App.vue
    │  main.js
    │
    ├─assets        // icons
    ├─components    // .vue
    └─store         // .js
```

Figure 4.2: The code structure of Module Sketch.

to modify a property, it has to emit the request back to its parent. For a complex system, this may lead to problems for maintenance, as it is hard to track which component did the modification. In this case, centralized state management will greatly simplify the task. Instead of passing the states down layer by layer, all components can get direct access to the states, as well as the mutations and actions to modify them. Thus, any state modification can be easily tracked. We use *Vuex* as our state management solution. More information about Vuex can be found in its documentation[2].

The structure of our centralized store is shown in Figure 4.4. In `index.js` we store the root states, which include the highly coupled states relevant to drafting features. Other states and their modification methods are extracted into modules.

4.3 Content Rendering

The entities in the work area are rendered from the objects in the local storage. An object contains all the information of the corresponding shape, such as its id, type, translation, and the coordinates of its vertices.

Rendering order is an important concept for a design tool. Imagine the user switches to select mode and clicks on a point where multiple shapes overlap. Intuitively the shape on the top, namely the latest drawn shape, would be selected. Thus, the content of the work area should be rendered by creating order (unless modified). So, we could assume that in the local storage, a list should exist to represent this order. On the other hand, as the user may need to

[2]https://vuex.vuejs.org/

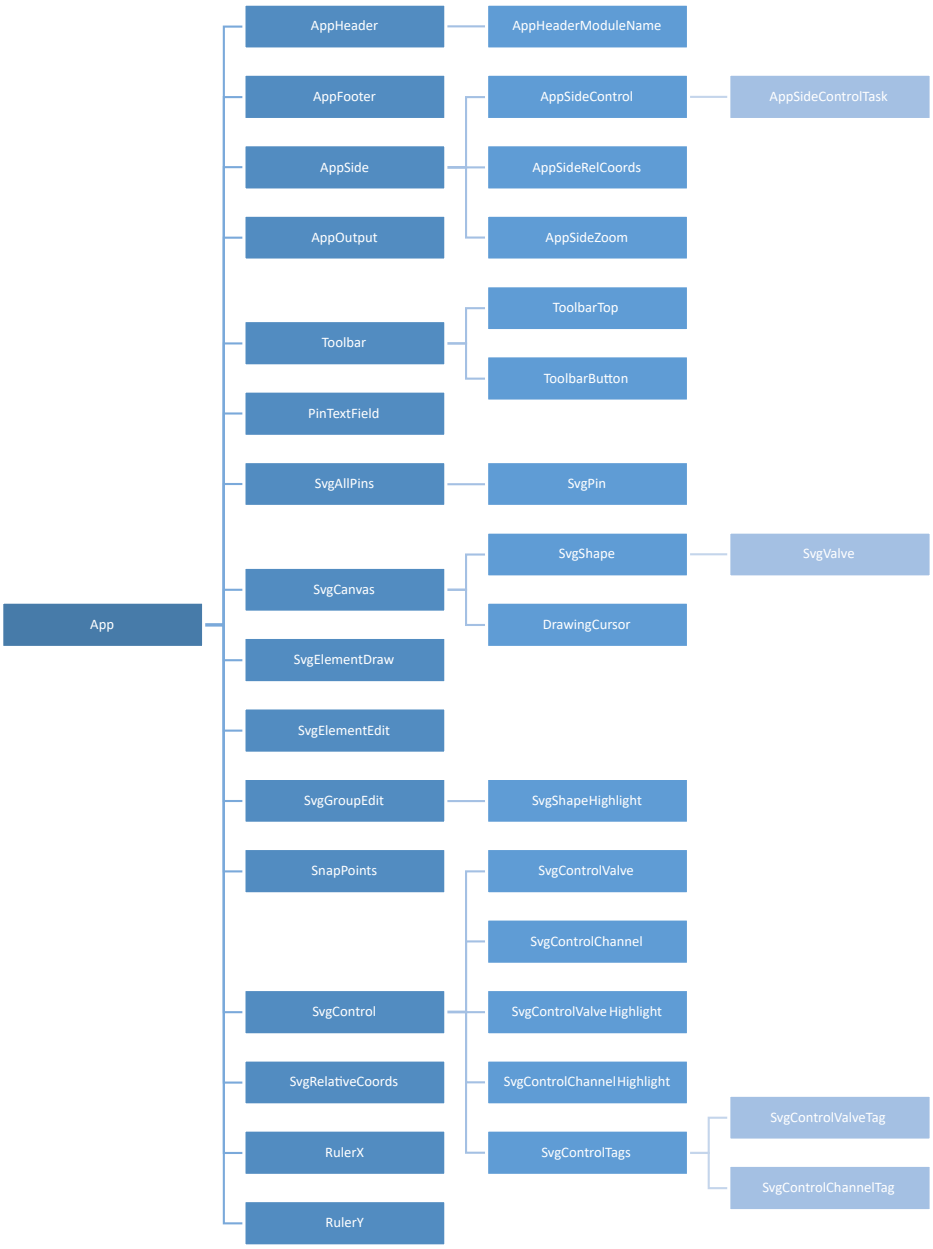


Figure 4.3: The component tree of Module Sketch.

```
store
   │  actions.js
   │  getters.js
   │  index.js
   │  mutations.js
   │
   └─modules
         controlBindings.js
         relativeCoords.js
         selectRect.js
         settings.js
         zoom.js
```

Figure 4.4: The structure of Module Sketch's centralized store.

select different shapes in the work area constantly, we should not iterate through the entire content list to find the correct shape every time, as it requires the time complexity of $O(n)$ and may result in deficient performance if the list is too long.

As shown in Listing 4.1, we use a Map (`contentObj`) to store all the objects of shapes in the work area. For each key-value pair, the key is the unique id of the shape, and all other information is stored in the value object. Although in JavaScript, the keys of a Map are ordered by insertion, we still need a separate list (`content`) to store the ids in rendering order because, in some cases, the user might modify the rendering order. For example, if one shape is entirely covered by another, the user may send the second shape to the bottom to make the first shape accessible.

Listing 4.1: Root states in index.js

```
// index.js
const state = () => ({
    crtLayer: 0,    //0: flow, 1: control
    contentObj: new Map(),  // map (id -> shape object)
    content: [[], []],   // only store ids
    pins: [[], []],   // only store ids
    ...
})
```

4.4 Drafting Implementation

As discussed in section 3.1, drafting is a complicated feature, and due to space constraints, we will selectively introduce its implementation.

4.4.1 Shape Drawing

We provide three drawing modes: (rounded) rectangle, ellipse, and polygon/polyline. Pin mode works in the same way as drawing a line section with an extra text entering step. After selecting the drawing mode, the user can add points to the work area to define a new shape.

Any shape will be defined by a certain number of points. For instance, each rectangle is defined by two points on one of its diagonals and each rounded rectangle needs one additional point to define the corner radius. A polyline can be defined by points of any number greater than one.

Mistakes are likely to occur during the drawing process, so the user should be able to execute the ***step out*** command at any time. It is, in our case, a right-click. After issuing this command, the program will check whether the currently drawn graphic is valid. Namely, if the graphic has a qualified number of points, it will stay in the work area. Otherwise, it will be discarded. The same applies to the polyline drawing process because it will not terminate automatically before the next right-click, as there is no upper limit on the number of points for polylines. In addition, users can close a polygon by moving the cursor close to the starting point, and clicking when the end point is snapped to it.

We depict the drawing process of Module Sketch in a state diagram (Figure 4.5). Listing 4.2 shows the pseudocode of the corresponding event handlers.

Listing 4.2: The pseudocode of the event handlers for the drawing process

```
handleMouseDownLeft():
    if not isAddingNewElm then
        isAddingNewElm <- True
        crtElm <- newElement()
    end if

handleMouseDownRight():
    if isAddingNewElm then
```

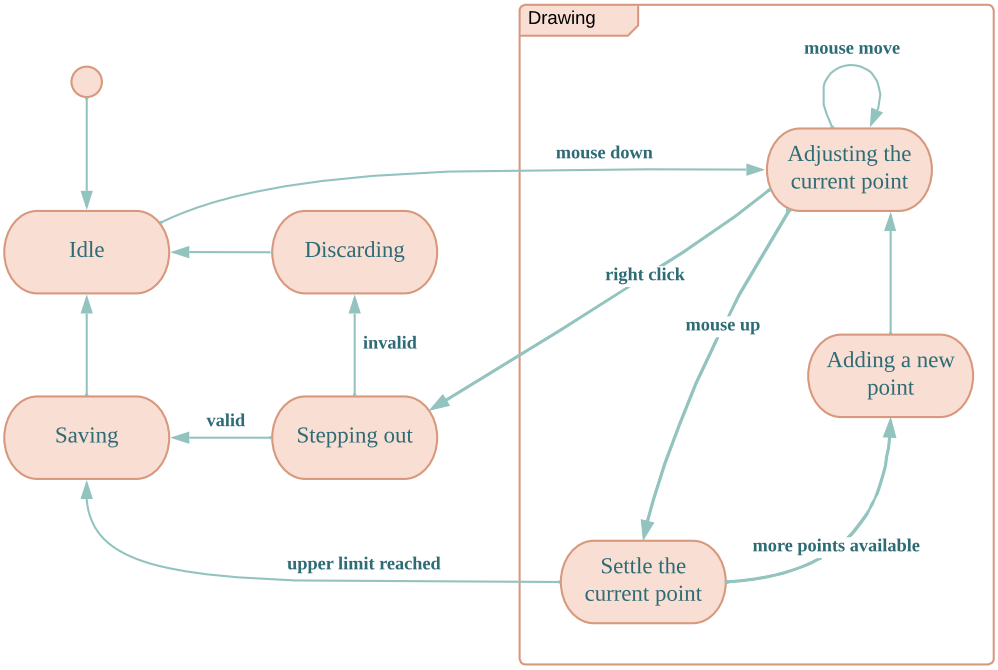


Figure 4.5: The state diagram for the drawing modes.

```
        if crtElm.points.length >= pointsLowerLimit[mode] then
            finishAddingElm()
        else
            discardCrtElm()
        end if
    end if

handleMouseUpLeft():
    if isAddingNewElm then
        crtElm.points.push(mousePosition)
        if crtElm.points.length == pointsUpperLimit[mode] then
            finishAddingElm()
        end if
    end if

handleMouseMove():
    if isAddingNewElm then
        crtElm.points[-1] <- mousePosition
    end if
```

4.4.2 Shape Selecting & Editing

In the select mode, the user can select shapes from the work area and edit them. In Figure 4.6 we introduce the selection process through a state diagram. The group selection can be achieved by a rectangle selection or clicking on multiple shapes with the Ctrl or Shift key down. The latter is represented by the transitions marked with $*$ in the figure. Listing 4.3 shows the pseudocode of the event handlers for the rectangle selection.

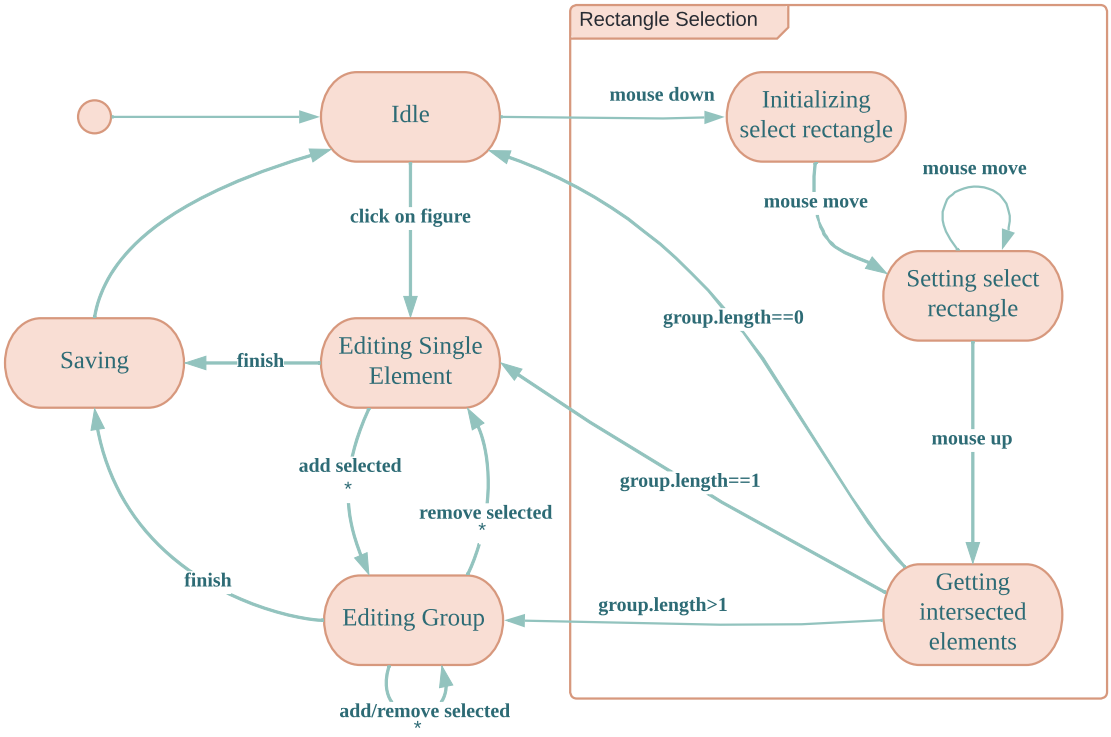


Figure 4.6: The state diagram for the select mode.

Listing 4.3: The pseudocode of the event handlers for the rectangle selection

```
# initiallize selecting rectangle at mouse position
handleMouseDownLeft(): initSelectRect()
# set the width and height of selecting rectangle
handleMouseMove(): setSelectRect()
handleMouseUpLeft():
    crtGroup <- []
    for each shape in current layer do
        if hasIntersection(shape, selectRect) then
```

```
        crtGroup.push(shape)
    end for
    if crtGroup.length == 0 then
        dicard()
    else if crtGroup.length == 1
        selectedShape <- crtGroup[0]
        startEditingElement()
    else
        selectedGroup <- crtGroup
        startEditingGroup()
    end if
```

After a valid selection, a `SvgElementEdit` or `ScgGroupEdit` component will be rendered accordingly. It will mount above the other shapes so they will not interfere with the editing process. Available editing options depend on the shape type, as shown in Table 4.1.

Table 4.1: Editing options for different shape categories.

Editing Option	Rectangle/Ellipse	Rounded Rectangle	Polyline	Polygon	Group
move	√	√	√	√	√
resize	√	√			
set fill	√	√		√	√
move vertex			√	√	
set corner radius		√			

4.4.3 Grid Size & Zooming

- **Grid and Flexible Precision**
  The workspace provides users with a background grid to aid in alignment. The grid size matches the smallest possible distance between two different points in the work area. It is dependent on the scaling with a range of 5μm-100μm. In other words, the further users zoom in, the more delicate shapes they can draw. While drawing or editing, the coordinate of the current point, which is being added or moved, is always rounded to the closest grid point.

- **Zoom-in/out and Panning**
  Zoom-in and zoom-out can be triggered by either the mouse wheel or clicking buttons. This feature is implemented in `zoom.js` and is achieved by adjusting the size of SVG

elements with the `viewbox` attribute fixed. The `height` and `width` of the `<svg>` are bound to `canvasSide`. Our zooming feature is animated using GSAP[3].

In `zoom.js` we provide a hard-coded list of doubles, which represents the number of pixels that ten μm is displayed on the work area (initialized with 4), and an integer that represents the index of the current value. The available scale range can be easily changed by modifying the list.

Listing 4.4: The module zoom.js

```
const state = () => ({
    // initially 1px represents 2.5 um
    globalTranslateX: window.innerWidth / 2 * 2.5, // value for gsap
    globalTranslateY: window.innerHeight / 2 * 2.5, // value for gsap
    globalTranslateXFinal: window.innerWidth / 2 * 2.5, // store end value
    globalTranslateYFinal: window.innerHeight / 2 * 2.5,// store end value
    zoomIdx: 8,
    gridSizeList: [1, 1.25, 1.5, 1.75, 2, 2.5,...],// 10 um displayed in px
    gridSize: 4, // value for gsap, end value = gridSizeList[zoomIdx]
    tenthViewBoxSize: 10000, // hard coded
})

const getters = {
    canvasSide(state) { return state.gridSize * state.tenthViewBoxSize }
    ...
}
const mutations = {
    // zoom-in with mouse wheel
    zoomInAtMouse(state, mousePosition) {
        if (state.zoomIdx + 1 < state.gridSizeList.length) {
            const [x, y] = zoomInTranslate(state, mousePosition)
            state.globalTranslateXFinal = x
            state.globalTranslateYFinal = y
            state.zoomIdx++
            gsap.to(state, { globalTranslateX: x, globalTranslateY: y,
                gridSize: state.gridSizeList[state.zoomIdx], duration: 0.25 })
        }
    },
    ...
}
```

[3]GSAP is a dependency used for JavaScript animation. For more information see: https://greensock.com/gsap/

4.5 Binding Options

Adding binding options is an independent functionality implemented in `controlBindings.js`. As stated in section 3.2, users can add binding options to single valves, as well as valve groups. Moreover, the valves and valve groups can be linked together as one binding-option-selection task (control task).

To start, the user needs to set the current layer to *Control* and switch to *bind* mode. Then the system will run an automatic check-up program to see if the existing valve groups and binding options are still well-defined. Undefined valves, which may occur, for example, when the user deletes the valve after assigning binding options to it, and channels will be removed. After that, users can add bindings, form groups, link existing groups, as well as edit and delete the bindings, groups, or links. Figure 4.7 shows the user interface while editing binding options.

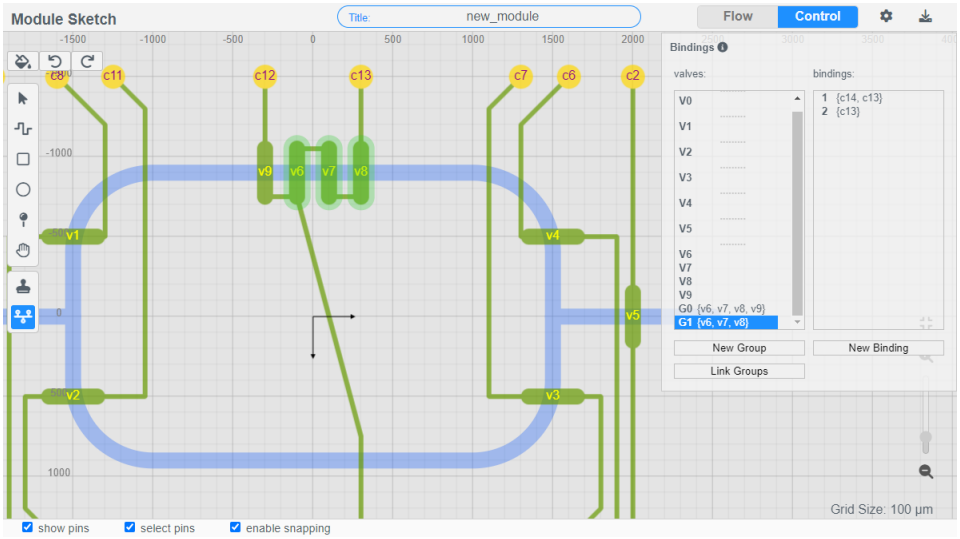


Figure 4.7: UI for editing binding options.

Now, the question is: how can we optimize this feature? We may notice that the collection of the valve set of each control task conforms to the **disjoint-set** data structure. To modify disjoint sets, we can use **union-find** algorithm.

4.5.1 Union-Find Algorithm

Union-find algorithm is an efficient approach to maintaining a partition of a set. In our case, the set is the union of all the valves and valve groups, and the partition stands for the disjoint valve sets of the control tasks.

Union-find algorithm has two core operations:

- `find(a)`: Find the *root* of the element `a`, namely the representative of the set (the predecessor of each element is initialized with itself).
- `union(a, b)`: Merge the set containing the element `a` with the set containing `b`.

Theoretically, the optimized operation of find or union on disjoint sets with n nodes takes time complexity $O(\alpha(n))$, where $\alpha(n)$ is the Inverse Ackermann function[4] .

Listing 4.5: Our implementation of the union-find algorithm

```
// initialize: foreach v in {valves and valve groups} do parent.set(v, v)
function find(v, parent) {
    // find the root of v
    let root = v
    while (root !== parent.get(root)) {
        root = parent.get(root)
    }
    // compress the path
    let tmp = v
    while (tmp !== root) {
        const nxt = parent.get(tmp)
        parent.set(tmp, root)
        tmp = nxt
    }
    return root
}
// merge the control tasks containing v1 or v2
function union(v1, v2, parent) {
    const [r1, r2] = [find(v1, parent), find(v2, parent)]
    if (r1 !== r2) parent.set(r2, r1)
}
```

[4]It grows extremly slowly and is very close to constant.

4.5.2 Valve Group Modification

When we form a new valve group, theoretically, the group and the single valves it contains should belong to the same control task, since one valve cannot be assigned to multiple control tasks. Thus, to form a new valve group through the selection, we need to merge all control tasks that contain at least one of the selected valves. The same goes for the editing of a valve group, and control tasks with newly added valves will be merged.

Listing 4.6: Confirm a valve group modification

```
// confirm adding or editing a valve group
function confirmModifyGroup(group_id, parent) {
    // initialize the predecessor if the group is new
    if (parent.get(group_id) === undefined) parent.set(group_id, group_id)
    // merge control tasks
    valveGroups.get(group_id).forEach(v => { union(group_id, v, parent) })
}
```

As long as the valve group exists, it maintains the dependencies on the valves it contains. When we delete a group, we will simply remove it without modifying the corresponding control task. However, if users want to release the old dependencies and split the control task, they can use the *unlink* operation specified in subsection 4.5.3.

Listing 4.7: Delete a valve group

```
function deleteGroup(group_id) {
    valveGroups.delete(group_id)
}
```

4.5.3 Link Modification

The link operation enables users to add additional dependencies between control tasks. These dependencies can be lifted if not bound by a valve group. Users can merge multiple control tasks together after selecting an arbitrary item from each control task.

Listing 4.8: Link the selected valves and valve groups

```
// confirm merging control tasks
function confirmLinking(selectedItems, parent) {
    if (selectedItems.length > 1) {
```

```
        selectedItems.forEach(i => { union(selectedItems[0], i, parent) })
    }
}
```

The unlink operation on a control task will break the dependencies that are not bound by an internal valve group through the crushing-and-reconstituting technique.

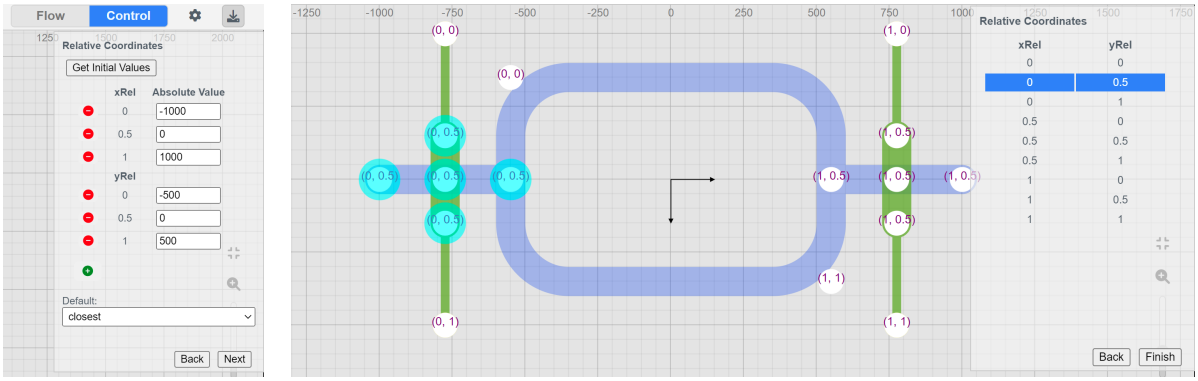
Listing 4.9: The unlink operation

```
const actions = {
    unlinkGroup({ commit, getters, state }, linkId) {
        const groups = [] //collect the ids of the valve groups in the task
        // crushing: reset the parent of each element to itself
        getters.linkedGroup.get(linkId).forEach(
            id => {
                if (id.includes('g')) groups.push(id)
                commit('resetParent', id)
            }
        )
        // reconstituting: rebuild valve groups in the task
        groups.forEach(gid => {
            state.valveGroups.get(gid).forEach(v => {
                // calls union(v1, v2, parent)
                commit('unionMutation', { v1: gid, v2: v })
            })
        })
    },
    ...
}
```

4.6 Design Interpretation

Before we get the final result, the design must be preprocessed by setting relative coordinates. It can be done through two simple steps shown in Figure 4.8.

First, we need to enter all the relative values in a form (4.8a). By default, three values will be given for both the x-axis and the y-axis, namely 0, 0.5, and 1. Users can also delete unnecessary values, as well as add new ones, then fill out the form with absolute values (as suggested by the rulers). To avoid potential mistakes, we recommend users to use the ***get***



(a) (b)

Figure 4.8: UI for assigning relative coordinates.

initial values function, which will iterate over the content and assign the absolute thresholds to relative values 0 and 1 (even if they are deleted). Intermediate values are then assigned by proportion. Before proceeding to the next step, the user can choose the algorithm for calculating the default relative coordinates, which gives either the closest or a fixed relative point.

In the second step(4.8b), the user can modify the default settings computed in the previous step. When all is settled, the user can click the *finish* button. It will tell the system to create and mount the Vue component `AppOutput`, where the result will be displayed (Figure 4.9). The functions of interpreting the design to the JSON format are implemented in `AppOutput.vue`, as they are not shared by other Vue components.

Figure 4.9: An example of the output.

5 Conclusion

In summary, we achieved our goal of developing a user-friendly design platform for microfluidic components. Module Sketch can simplify the design process of microfluidic IP modules, and support the module library expansion for design automation software.

Although, in practice, the roles of UI/UX designers and developers are usually separated, we put a lot of thought into analysing, designing, and optimizing the interactivities between the user and our application. This process provides us with the foundation and framework for our implementation, upon which the comprehensive and flexible features of drafting, describing binding options, and design interpretation are implemented.

In addition, Module Sketch is a pure front-end application with no back-end implementation involved, which means that once the page is loaded, there will be no more interactions between the server and the browser. Everything will be handled locally on the client-side. As for the compatibility, we developed Module Sketch on Google Chrome and tested it on Firefox and Microsoft Edge, making it fully supported by the mainstream browsers.

6 Future Work

In this chapter, we will discuss the vision for potential future work from our own view.

6.1 Load Existing Designs

Module Sketch currently only allows users to design mLSI components from scratch, namely, users cannot load existing designs and edit them. In the future, we might enable the user to continue the previous work by uploading the existing design of .JSON and parsing it into the work area.

6.2 Netlist Recognition

The idea is to bind some figures or some vertices in the work area together so that the user can edit the whole group conveniently and avoid misediting on figures that are dependent on the group. However, what we expect is not simply to form groups by selecting figures, but the automatic recognition of the netlist. It could be an extension of snapping: when the user operates with the snap point, the relevant figures will be bound, and if one of the figures is moved, the position of the others will be modified accordingly. This is only an idea at present, and the specific mechanism has yet to be designed, but it may be useful in some scenarios.

6.3 State Management Migration

During the developing process, we used Vuex 4 as the solution for our centralized store, but recently the official state management library for Vue projects was changed to Pinia.

Although Vuex 4 is still maintained, new features may not be added to it. Thus, migrating to Pinia might be an appropriate choice.

6.4 Drafting Feature Extension

As Columba is still in the development stage, it is likely that there will be new requirements that Columba needs to meet. For example, the channel width for Columba is currently fixed, but it may need to support customized channel width in the future. In such cases, Module Sketch needs to make corresponding adjustments. Another potential reinforcement is the function of drawing arcs. Although we have implemented the triangle circumscribed circle algorithm to define the arc with three points, this function was eventually deprecated due to a lack of flexibility. There is a possibility that this feature will be achieved by other means in the future. For instance, a function of adding rounded corners to polylines would be an option to consider.

List of Figures

List of Tables

Listings

Bibliography

[1] T. Thorsen, S. J. Maerkl, and S. R. Quake. "Microfluidic Large-Scale Integration". In: *Science* 298.5593 (2002), pp. 580–584. DOI: 10.1126/science.1076996. eprint: https://www.science.org/doi/pdf/10.1126/science.1076996. URL: https://www.science.org/doi/abs/10.1126/science.1076996.

[2] G. M. Whitesides. "The origins and the future of microfluidics". In: *Nature* 442 (2006), pp. 368–373. DOI: 10.1038/nature05058.

[3] R. Daw and J. Finkelstein. "Lab on a chip". In: *Nature* 442 (2006), pp. 367–367. DOI: 10.1038/442367a.

[4] T.-M. Tseng, M. Li, D. N. Freitas, T. McAuley, B. Li, T.-Y. Ho, I. E. Araci, and U. Schlichtmann. "Columba 2.0: A Co-Layout Synthesis Tool for Continuous-Flow Microfluidic Biochips". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.8 (2018), pp. 1588–1601. DOI: 10.1109/TCAD.2017.2760628.

[5] E. E. Tsur. "Computer-Aided Design of Microfluidic Circuits". In: *Annual Review of Biomedical Engineering* 22.1 (2020). PMID: 32343907, pp. 285–307. DOI: 10.1146/annurev-bioeng-082219-033358. eprint: https://doi.org/10.1146/annurev-bioeng-082219-033358. URL: https://doi.org/10.1146/annurev-bioeng-082219-033358.

[6] T.-M. Tseng, M. Li, B. Li, T.-Y. Ho, and U. Schlichtmann. "Columba: Co-layout synthesis for continuous-flow microfluidic biochips". In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2016, pp. 1–6. DOI: 10.1145/2897937.2897997.

[7] T.-M. Tseng, M. Li, Y. Zhang, T.-Y. Ho, and U. Schlichtmann. "Cloud Columba: Accessible Design Automation Platform for Production and Inspiration: Invited Paper". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–6. DOI: 10.1109/ICCAD45719.2019.8942104.

[8] Y. Zhang, T.-M. Tseng, and U. Schlichtmann. "Portable all-in-one automated microfluidic system (PAMICON) with 3D-printed chip using novel fluid control mechanism". In: *Scientific Reports* 11 (2021), pp. 1–10. DOI: 10.1038/s41598-021-98655-9.

[9] T.-M. Tseng, B. Li, U. Schlichtmann, and T.-Y. Ho. "Storage and Caching: Synthesis of Flow-Based Microfluidic Biochips". In: *IEEE Design & Test* 32.6 (2015), pp. 69–75. DOI: 10.1109/MDAT.2015.2492473.

[10] WHATWG. *HTML Standard*. Last accessed 26 June 2022. 2022. URL: https://html.spec.whatwg.org/multipage/.

[11] T. J. DeGroat. *The History of JavaScript: Everything You Need to Know*. Last accessed 26 June 2022. 2019. URL: https://www.springboard.com/blog/data-science/history-of-javascript/.

[12] J. J. Garrett et al. "Ajax: A new approach to web applications". In: (2005). Last accessed 26 June 2022. URL: https://www.scriptol.fr/ajax/ajax_adaptive_path.pdf.

[13] MDN. *SVG: Scalable Vector Graphics*. Last accessed 26 June 2022. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/SVG.

[14] MDN. *A re-introduction to JavaScript*. Last accessed 26 June 2022. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript.

[15] Node.js. *A brief history of Node.js*. Last accessed 26 June 2022. 2022. URL: https://nodejs.dev/learn/a-brief-history-of-nodejs.

[16] SlashData. *State of the Developer Nation 22nd Edition*. Last accessed 26 June 2022. 2022. URL: https://www.developernation.net/resources/reports/state-of-the-developer-nation-q1-2022.

[17] GitHub. *The 2021 State of the Octoverse*. Last accessed 26 June 2022. 2022. URL: https://octoverse.github.com/.

[18] S. Greif et al. *The 2021 State of JS*. Last accessed 26 June 2022. 2022. URL: https://2021.stateofjs.com/en-US/.

[19] F. Hámori. *The History of React.js on a Timeline*. Last accessed 26 June 2022. 2022. URL: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/.

[20] *Angular Docs*. Last accessed 26 June 2022. 2022. URL: https://angular.io/docs.

[21] V. Cromwell. *Evan You*. Archived from betweenthewires.org on June 3, 2017. Last accessed 26 June 2022. 2016. URL: https://web.archive.org/web/20170603052649/https://betweenthewires.org/2016/11/03/evan-you/.

[22] *Vue.js Docs*. Last accessed 26 June 2022. 2022. URL: https://vuejs.org/guide/introduction.html.

[23] T.-M. Tseng, M. Li, D. N. Freitas, A. Mongersun, I. E. Araci, T.-Y. Ho, and U. Schlichtmann. "Columba S: A Scalable Co-Layout Design Automation Tool for Microfluidic Large-Scale Integration". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465905.

[24] R. Sanka, J. Lippai, D. Samarasekera, S. Nemsick, and D. Densmore. "3DμF - Interactive Design Environment for Continuous Flow Microfluidic Devices". In: *Scientific Reports* 9 (2019), pp. 1–10. DOI: 10.1038/s41598-019-45623-z.

[25] N. Amin, W. Thies, and S. Amarasinghe. "Computer-aided design for microfluidic chips based on multilayer soft lithography". In: *Proceedings of the 2009 IEEE International Conference on Computer design* (2009), pp. 2–9.