IMPERIAL

Evaluation and Re-implementation of Issie on the .NET Platform

Author

Kaiwen Liu

CID: 01845986

Supervised by

Dr Thomas Clarke

Second Marker

Mr S. Baig

A Thesis submitted in fulfillment of requirements for the degree of
Master of Engineering in Electronic and Information Engineering

Department of Electrical and Electronic Engineering
Imperial College London
2024

Abstract

This project aims to re-implement and evaluate the Interactive Schematic Simulator with Integrated Editor (Issie)[1], a digital circuit simulation application used at Imperial College London, on the .NET desktop platform[2] to enhance efficiency, security, and maintainability. The report provides implementation details, future works and evaluates the necessity of making a full port to a new platform based on performance comparison.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced.

I have used ChatGPT v4 as an aid in the preparation of my report. I have used it to improve the quality of my English throughout, however all technical content and references comes from my original text.

Copyright Declaration

Acknowledgments

I would like to express my deepest gratitude to my tutor, Dr. Thomas J. W. Clarke of Imperial College London, for his invaluable guidance, patience, and expert advice throughout the duration of this project. His insights and suggestions have been crucial in shaping both the direction and the success of this work.

I am also immensely thankful to my friends and family for their unwavering support and encouragement. Their belief in my abilities and constant motivation has been a source of strength and inspiration during the challenges of this research.

Their collective support has been instrumental in my personal and academic growth, and I am deeply appreciative of their contributions to my journey.

Contents

List of Acronyms

Issie	Interactive Schematic Simulator with Integrated Editor
UI	User Interface
GUI	Graphic User Interface
MVU	Model-View-Update
API	Application Programming Interface
DOM	Document Object Mode
CPU	Central Processing Unit
EEE	Electrical & Electronic Engineering
FRP	Functional Reactive Programming
DSL	Domain Specific Language
IDE	Integrated Development Environment
IPC	Inter-process Communication
NPM	Node Package Management
JIT	Just in Time

List of Figures

1 Introduction

Issie, designed for digital circuit design and simulation, targets students and hobbyists eager to learn Digital Electronics in a user-friendly way. Developed at Imperial College London, Issie leverages F#[3] for its core functionality, creating a cross-platform application within the Web ecosystem.

The tech stack of Issie is complex, combining web and .NET technologies which can be challenging for new developers to grasp. It primarily uses F# with an Model-View-Update (MVU) [4] architecture, trans-compile via Fable[5] to utilize Electron.JS[6] creating a cross-platform desktop application that makes system-level interaction.

Transitioning to a .NET cross-platform framework could enhance Issie by streamlining the toolchain and boosting performance and maintainability.

This project aims to assess the current Issie tech stack and .NET ecosystem alternatives, identifying a suitable Graphic User Interface (GUI) framework and Domain Specific Language (DSL) on .NET that preserves Issie's essential functionalities and coding style. We will execute a structured port to this new technology stack, noting platform differences and challenges encountered during the transition. The project will culminate in a comprehensive performance evaluation to analyze the differences and determine the feasibility and value of porting an application with 42k lines of code.

2 Background Research

Contents

To lay the groundwork for this project's exploration and transformation, we begin with an overview of Issie's current functionalities and technology stack. This introduction highlights the existing framework's strengths and limitations, clarifying why we are considering transitioning to a new stack. This sets the stage for understanding the potential enhancements and innovations a new GUI framework could offer Issie.

2.1 Issie

Issie is designed to be beginner-friendly and guide users toward their goals via clear UI signposting, error messages that explain how to correct errors, and visual clues. It is intuitive to use with small single-sheet circuits and also highly productive in debugging complex Central Processing Unit (CPU) designs made from 50 sheets and hundreds of thousands of components. The motivation for Issie was the observation that during digital lab work in Electrical & Electronic Engineering (EEE) courses, previously, more time was spent learning to use (or cope with subtle bugs in) large commercial hardware design tools than learning digital design. [1]

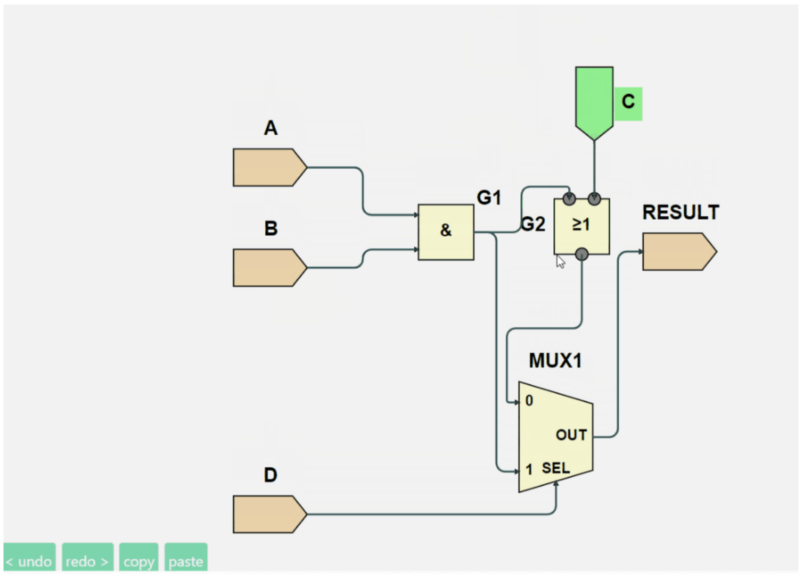
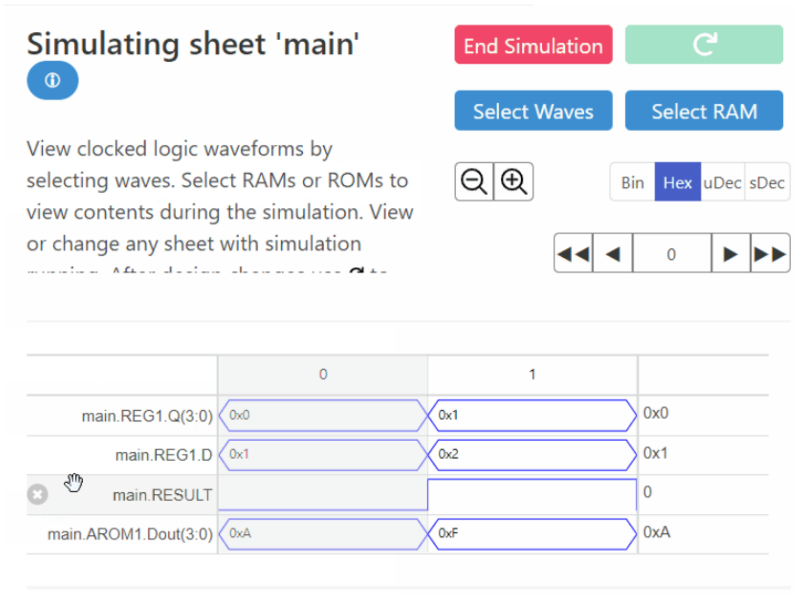


Figure 2.1: Schematic Circuit Editor. Image credits:[1].



Simulating sheet 'main'

View clocked logic waveforms by selecting waves. Select RAMs or ROMs to view contents during the simulation. View or change any sheet with simulation

	0	1	
main.REG1.Q(3:0)	0x0	0x1	0x0
main.REG1.D	0x1	0x2	0x1
main.RESULT			0
main.AROM1.Dout(3:0)	0xA	0xF	0xA

Figure 2.2: Waveform Simulation. Image credits:[1].

Issie is predominantly coded in F#, utilizing an MVU architecture derived from the Elmish[7] library. This F# code is subsequently transpiled into JavaScript[8] using the Fable compiler. Following this, Electron.JS is employed to transform the resulting web application into a cross-platform desktop application, and it enables the application to interact with platform-specific Application Programming Interface (API), such as file system access.

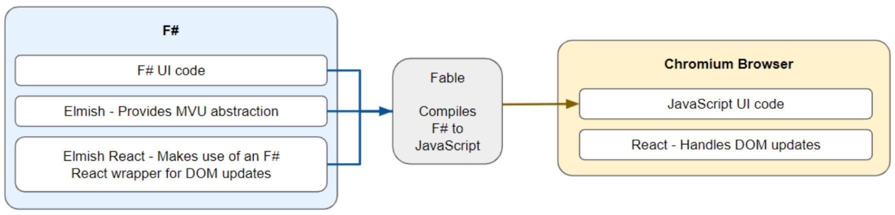

F#
F# UI code
Elmish - Provides MVU abstraction
Elmish React - Makes use of an F# React wrapper for DOM updates
Fable
Compiles F# to JavaScript
Chromium Browser
JavaScript UI code
React - Handles DOM updates

Figure 2.3: The Compilation Structure of Issie. Image credits[9].

The selection of this particular technology stack for Issie is driven by various factors. Primarily, the well-established nature of web ecosystems offers access to robust libraries and consistent support for multi-platform compatibility. Additionally, using a statically typed functional programming language offers significant advantages in state management and maintenance. The subsequent sections will delve into a detailed explanation and justification for the choice of web ecosystem 2.1.1 combining the MVU paradigm 2.1.3 in Issie's development.

2.1.1 Web Ecosystem

Issie's core architecture leverages the versatile Electron.JS framework, a Node-based platform inherently designed for multi-platform compatibility. This architecture choice provides several key benefits:

- **Cross-Platform Compatibility[10]**: Electron.JS enables Issie to operate seamlessly across various operating systems including Windows, Linux, and MacOS, which is crucial for its diverse user base of students and academic staff.
- **Open-Source Foundation**: Electron is open-source under the MIT license, aligning well with Issie's distribution under the GPLv3[11] license. This facilitates free usage and contributes to a community-driven development approach.
- **Popularity**: Electron's widespread adoption and recognition ensure a robust support community and extensive documentation, making it easier to maintain and enhance the application over time.
- **Local Execution Advantages**: Unlike standard web applications confined to browser capabilities, Electron allows Issie to leverage deeper system integration and richer functionalities, enhancing the overall user experience while maintaining web-like agility.

These attributes of the Electron.JS framework and the web-based ecosystem not only fulfill but enhance Issie's operational requirements, ensuring robust performance and broad accessibility.

In Issie, Electron architecture divides into two primary processes: the main program and the renderer. The main program handles startup procedures and engages in low-level machine I/O operations utilizing Node.JS[12] and native APIs. In contrast, the renderer, powered by Chromium[13], takes charge of the application's core functionalities and user interface, including crucial tasks like simulation and component rendering. This bifurcation facilitates a cleaner separation of concerns within the application architecture, offering distinct advantages in managing application logic. However, this complexity can also pose challenges, which we will explore in depth in the subsequent chapter 4.

Furthermore, Electron's architecture inherently requires significant memory resources, which can impact performance. Additionally, JavaScript's non-strict type system[14] often complicates the development process, pointing to the potential benefits of integrating a more strictly typed programming language. These considerations set the stage for the next section 2.1.2, where we discuss enhancements to address these limitations.

2.1.2 Functional Programming

F# is a strongly typed Functional programming language and was chosen as the main programming language for Issie for its great features to build robust and maintainable code such as:

Understanding and Maintenance F#'s functional nature, where functions transform immutable data, simplifies the maintenance process. This is particularly advantageous over object-oriented code, as it reduces concerns about state management. Refactoring is often as straightforward as altering types and adhering to compiler guidance for necessary code adjustments.[15]

Static type The F# compiler rigorously enforces type correctness and thorough consideration of all scenarios in pattern matching. This significantly reduces the likelihood of bugs. Consequently, code written in F# tends to function correctly on the first try, effectively counterbalancing the limitations encountered with JavaScript in Web ecosystem.[16]

Testing Efficiency Testing in F# is straightforward. Functions are tested by inputting data and verifying if the output aligns with the expected results. This process eliminates the need to manipulate the system into specific states for testing, streamlining the development workflow.[17]

Educational Alignment F# serves as the language of choice for high-level programming courses at Imperial College. This allows students to continuously contribute to the project in coursework and projects, meanwhile practicing their skills in functional programming, software development, and circuit simulation.

To seamlessly integrate F# into the web-based ecosystem of Electron, selected for Issie, the Fable compiler plays a pivotal role. Fable effectively bridges the gap between F# and the web environment by converting F# code into JavaScript. This transpilation allows the F# codebase to be compiled as a multi-platform application within the Electron framework, thereby leveraging the advantages of both F# and web technologies.[18]

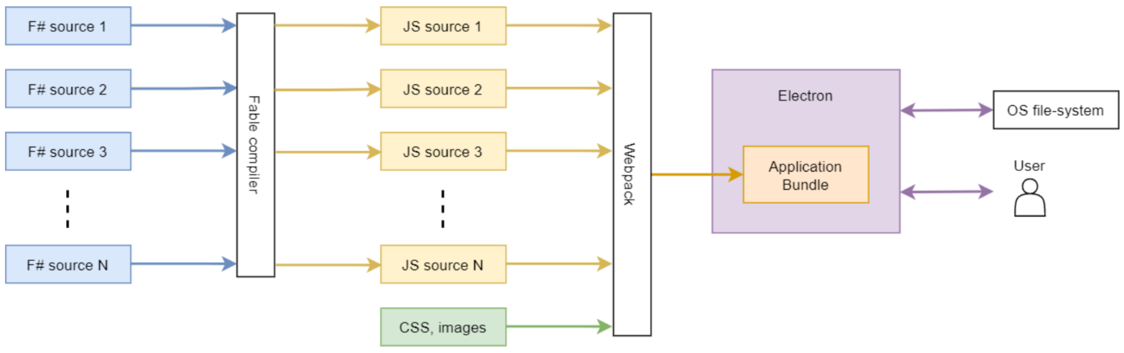


Figure 2.4: Fable Electron Application. Image credits:[9].

The transcompilation process depicted converts F# code into an Electron application by first using the Fable compiler to transpile the F# source files into JavaScript. These JavaScript files, along with associated assets like CSS and images, are then bundled together using Webpack, a module bundler that optimizes and packages web assets. The bundled code is integrated into the Electron framework, which allows for cross-platform desktop application development using web technologies. This Electron application can then interact with the operating system's file system and be distributed to users as a native desktop application.

2.1.3 MVU Architecture

Utilizing the F# programming language in the development of user interfaces is greatly enhanced by leveraging the Elmish library. Elmish empowers developers to harness the capabilities of Functional Reactive Programming (FRP)[19], a paradigm well-suited for UI design and creation. Inspired by the Elm programming language, Elmish adheres to the core FRP concepts but extends them within the context of F#, a language known for its power and versatility.

At the heart of Elmish is the Model-View-Update (MVU) architecture, a pattern that is a great practice of FRP paradigm and particularly effective for structuring user interfaces in applications and has gained popularity in the realm of functional programming. Originally popularized by the Elm language for web applications, MVU has been effectively adapted for F# through libraries like Elmish. This architecture facilitates the development of intuitive and responsive UIs by maintaining a clear separation of concerns between the model (data), the view (UI representation), and the update (state changes), making it an ideal choice for structuring user interfaces in F# applications.

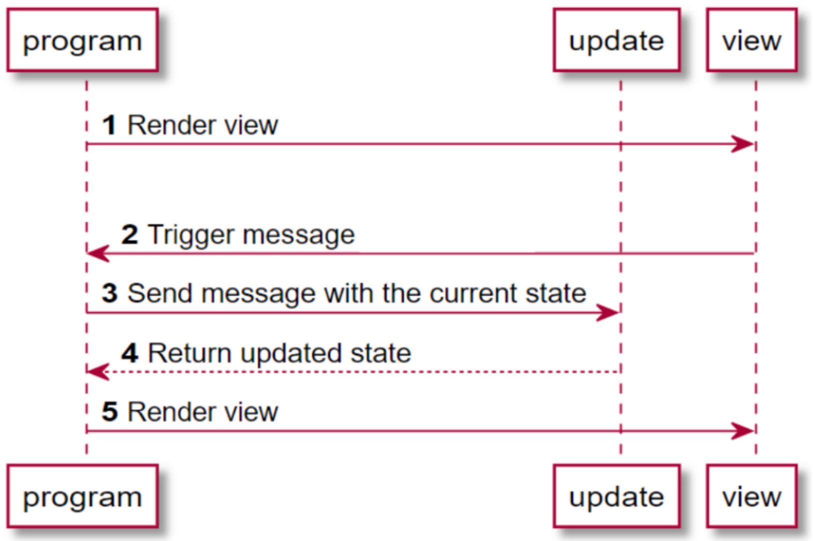


Figure 2.5: MVU Interactions in the Elmish Architecture. Image credits:[7].

- The model is defined as an F# record. It encapsulates the application's state.
- The view function uses the model to create a virtual Document Object Mode (DOM), which Elmish renders to the actual DOM in web applications or to the GUI in desktop applications. The dispatch function is used to send messages to the update function based on user interactions or other events
- The update function takes a message and the current model, then returns a new model. Elmish ensures that this function is pure – it does not mutate the model directly but instead returns a new instance of the model with any updates applied.

In Issie, more than 60 modules are implemented as a single component Elmish MVU application. This is the simplest type of Elmish application where all state is contained in a single Model type and there are no separate React[20] components with their own internal state. Although Elmish uses a React DOM (React has a virtual DOM that makes updates more efficient) developers can usually ignore React and write UI functions as though the DOM was the equivalent HTML[21].

Functional Component

Despite Fable.Elmish's excellent encapsulation of React components, there are still some complex aspects to understand about the Issie rendering process, particularly regarding wire and symbol component rendering. Functional components are introduced [22], which differ from the pure *View* function mentioned earlier. Unlike the Virtual DOM, which compares and updates every time, functional components persist within the Virtual DOM and only update when the properties of the function change. This design ensures very efficient rendering by skipping unchanged components in the DOM.

The efficient use of React also brings the primary challenge in transitioning Issie to a .NET cross-platform desktop UI, which requires achieving efficient SVG rendering for schematics while caching static objects without compromising the efficacy of the MVU code structure that has proven highly effective.

2.2 The .NET Re-implementations

The sections above outline the tech stack used by Issie, and they have been well The sections above detail Issie's well-established tech stack, which has been effectively implemented and tested over the years. Despite its proven reliability and user-friendliness, there is potential for further enhancement.

.NET, an open-source platform developed by Microsoft, supports a variety of programming languages including C#[23], F#, and Visual Basic[24], and caters to diverse applications across web, mobile, desktop, gaming, and IoT platforms. Transitioning Issie to a modern cross-platform desktop UI with .NET could address existing limitations and significantly enhance its capabilities.

Criterion	Web Ecosystem (Electron)	.NET Platform
Development Languages	JavaScript, HTML, CSS	C#, F#, VB
Cross-platform	Yes	Yes
Tool Chain	Complex	Simple
Community Support	Excellent	Excellent
Resource Usage	High	Moderate

Simpler Tool Chain Transitioning from a Fable-Electron trans-compile project to Avalonia simplifies the development and debugging toolchain by offering a unified .NET-centric environment, streamlining both the build process and debugging. This move eliminates the complexity of navigating between F#, JavaScript, and Electron, allowing direct use of .NET tools, libraries, and Integrated Development Environment (IDE) without the overhead of transpilation. Overall, adopting Avalonia fosters a more efficient, straightforward development process, fully integrated into the .NET ecosystem.

Better Performance As a native implementation of desktop applications, .NET is known for its performance, compared to the Electron-based application it should bring a certain level of improvement, by leveraging the .NET runtime's optimizations, reducing the application's resource consumption compared to the Electron's embedded Chromium and Node.js runtime [25].

2.3 Summary

In this background section, we have explored Issie's existing architecture and assessed its strengths and weaknesses, pinpointing issues such as a complex toolchain, significant memory consumption, and various performance challenges. We recognized the benefits of maintaining the MVU structure and using the F# language, which we intend to preserve in our forthcoming requirements section 3. These insights justify our decision to transition to a fully .NET-based technology stack, aiming to address and improve these critical areas. In the analysis section 4, we will discuss potential solutions to overcome these drawbacks effectively.

3 Requirement Capture

Contents

This chapter outlines the essential requirements for the re-implementation, focusing on selecting a new technology stack and preserving core functionalities. Initially, we must identify a tech stack that can deliver the benefits outlined in the previous section 2.2. Furthermore, we aim to comprehensively catalog Issie's existing functionalities to ensure their continuation and effective implementation in the new system.

3.1 Technical Requirements

As an open-source project designed for developers, it is imperative to maintain a coding style and logic that are consistent with the original, while ensuring the new technology meets or exceeds the advantages of the existing Web ecosystem 2.1.1. Hence, the new tech stack should fulfill the following requirements:

- **Tech Stack Compatibility:** It is crucial to preserve our use of functional programming and the MVU architecture to ensure seamless transition and maintainability.
- **Performance with SVG Rendering:** The chosen UI framework must be capable of efficiently managing SVG rendering. The focus should be on performance enhancements,

particularly through effective object caching in the schematic editor.

- **Cross-Platform Compatibility:** Ensuring consistent functionality across various operating systems is essential for enhancing Issie's usability and user reach.
- **Developer Ecosystem and Support:** A strong and growing developer community is vital. The new technology should support a modern and flexible architecture to facilitate easier maintenance and troubleshooting.

3.2 Functionality Requirement

We can divide Issie's specific functionalities into different modules and then assess the necessity and difficulty of the transition.

Table 3.1: Issie Module Porting Difficulty Assessment

Module	**Description**	**Difficulty**	**Importance**
DrawBlock	Circuit rendering and interaction	High	High
Simulator	Simulation Logic and Interface	Medium	High
File IO	File interaction with saved sheets	Minimal	High
General Interface	Various UI elements in application	Medium	Medium

After accessing each module of Issie, there are two crucial modules required for a systematic evaluation of the new tech stack while requires a significant amount of work, which are *DrawBlock* and *Simulator*:

The DrawBlock module works on all the interactions with sheets, wire, and symbols, providing the most crucial functionality of Issie as a CAD application and a key performance evaluation factor in the rendering aspect.

The simulator consists of mostly logical calculation with minimal interaction with GUI, it should be easy and crucial for porting since it provides a core evaluation metric, the simulation speed. The step-wise simulation function is another primary choice of porting to achieve a complete move deliverable and straightforward representation of simulation.

Combining basic menu and popup GUI, file access and management, we now have a full list of requirements of functionality at this stage below:

Table 3.2: Functionality Requirement Summary

Module	**Block**	**Functionality**	**Category**
Drawblock	**Sheet**	Render circuits	Rendering
		Zoom canvas	Interactive
		Drag canvas	Interactive
	Wire	Render wire with different widths and labels	Rendering
		Automatic routing	Arithmetic
		Connect/disconnect from port	Interactive
	Symbol	Render all Issie symbol types	Rendering
		Move symbols	Interactive
		Zoom and rotate symbols	Interactive
		Copy symbols	Interactive
		Multiple symbol selection	Interactive
		Overlap detection	Arithmetic
Simulation	**Simulator**	Read in circuit	SystemIO
		Step/Waveform output	Arithmetic
	Step Simulation	Basic display of step, input, output	Rendering
		Input/Step Modification	Interactive
		Save/Load Simulation Config	SystemIO
	Waveform Simulation	Waveform Render	Rendering
		Wave Selection	Interactive
		RAM selection	Interactive
FileIO	**Project**	Read in demo project	SystemIO
		Read in legacy project	SystemIO
		Save project	SystemIO
	Sheet	Import Sheet	SystemIO
		Copy/Paste Sheet	SystemIO
General Interface	**MainView**	Top Menu for project management	Interactive
		Catalogue menu for symbol creation	Interactive

Continued on next page

Module	**Block**	**Functionality**	**Category**
		Simulation menu	Interactive
	PopUP	Project Popup window when app start	Interactive
		Notifications	Rendering
		Simulation Related Popup	Interactive

There are four main categories of functionality that need to be implemented, with interactive features and rendering constituting the primary workload of the porting process. File I/O will also require some time to adapt, while arithmetic code blocks should be directly portable. A detailed process and solutions for each category are discussed in sections 5.2 and 5.4.

3.3 Conclusion

The requirements outlined serve as both a blueprint for the deliverables expected from the project and a guide for the implementation process detailed in Chapter 5. Additionally, they provide crucial context for understanding the design decisions discussed in Chapter 4.

4 Analysis and Design

Contents

Building on the established technical and functional requirements in Chapter 3, we are set to select an appropriate GUI framework from the .NET ecosystem. We will then analyze its characteristics and compare them with those of the previous setup to gauge potential performance improvements. This will help set the stage for further detailed test and evaluation.

4.1 Availability of GUI Frameworks

In transitioning to the .NET ecosystem while maintaining the MVU structure and the use of F#, several factors need to be considered. These factors are detailed in the previously listed requirements (see Section 3.1).

Among the potential candidates for .NET cross-platform frameworks are QTSharp[26], WPF[27], and Avalonia[28]. These frameworks are generally designed for multi-platform support and perform SVG rendering using a XAML[29]-based approach. The primary aspects to evaluate include compatibility with F# and the MVU architecture, alongside other criteria such as popularity, openness, and cross-platform capabilities:

Framework	F# Support	MVU Support	Open Source	Cross-Platform
Avalonia	✓	✓	✓	✓
WPF	✓	✓	✓	✗
UNO Platform	✗	✗	✓	✓
.NET MAUI	✓	✓	✓	✓
QtSharp	✗	✗	✓	✓

Table 4.1: Evaluation of .NET Cross-Platform Frameworks

Avalonia: Avalonia is increasingly favored in the F# community due to its compatibility with F# principles[30]. It supports the MVU architecture, especially when used with libraries like Avalonia.FuncUI[31] or Fabulous.Avalonia[32], enabling straightforward implementation of MVU patterns in F# applications.

WPF (Windows Presentation Foundation): While WPF can be used with F#, it is primarily designed for C#/XAML, making its integration with F# less intuitive, implementing MVU architecture using Elmish.WPF[33] is possible but not widely adopted. Its primary limitation is its exclusive support for the Windows platform[34].

Uno Platform[35]: The Uno Platform is versatile, offering cross-platform capabilities with exceptional mobile support[36]. However, its support for F# is somewhat basic, making it less ideal for projects that require deep integration with F# or extensive use of the MVU pattern.

MAUI (Multi-platform App UI)[37]: Developed by Microsoft, MAUI is an evolution of Xamarin.Forms and a key part of the .NET 6 initiative[38]. It aims to provide a unified framework for building cross-platform applications but is still more specialized in mobile development. While it supports F#, it's still maturing regarding its integration with F# programming and MVU patterns.[39]

QtSharp: QtSharp bridges the Qt application framework with the .NET environment, predominantly targeting C# for .NET bindings. Due to its roots in C++, it's less aligned with F# programming paradigms and doesn't inherently support the MVU pattern. This makes QtSharp more suitable for projects prioritizing C++ and .NET interoperability over native F# development.

Based on the comparison above, only Avalonia and MAUI provide good support for F# and MVU structure while providing cross-platform ability.

Avalonia UI and MAUI represent two distinct approaches to cross-platform GUI framework implementation. Avalonia employs a custom rendering engine that ensures uniformity in UI elements across all platforms [40]. This consistency is pivotal for applications that require a standardized user experience regardless of the operating system. In contrast, MAUI leans heavily towards mobile platforms and utilizes Blazor for Linux support, which is not inherently part of its core framework but rather a community-driven extension [41].

Given these differences, Avalonia UI is the preferable choice for our needs. Its comprehensive platform support, including Linux, which is essential for Issie, along with its strong and active community [42], positions it as the more suitable and reliable framework. Avalonia's consistent rendering engine promises a seamless user experience across desktop environments, which is a critical requirement for our project. Conversely, MAUI's primary focus on mobile platforms and reliance on community solutions for Linux support may not align as effectively with our project's goals for a cross-platform desktop application.

4.1.1 DSL and Wrapper

There are two approaches to implementing the MVU structure on Avalonia, Avalonia.FuncUI[43], and Fabulous.Avalonia[32], both libraries that facilitate the use of the MVU architecture in Avalonia applications as wrapper layer.

FuncUI is a library designed to build Avalonia UIs using a functional approach. It offers a DSL[44] for defining UI elements in F#. FuncUI provides a more direct and functional way of defining UI elements, aiming to keep everything within F#. It allows for a concise and type-safe definition of UI, leveraging F#'s strong typing system.

Fabulous provides MVU wrapper support for various .NET GUI frameworks including Avalonia and MAUI, similar to FuncUI it made a custom DSL which the following section will make a comparison on.

DSL comparison:

Per the specifications in Section 3.1, the new implementation must retain a similar MVU structure to the original Issie to facilitate a straightforward and efficient porting process. To evaluate how each platform aligns with this requirement, we will compare their MVU implementations and DSL using the button of a simple counter app example:

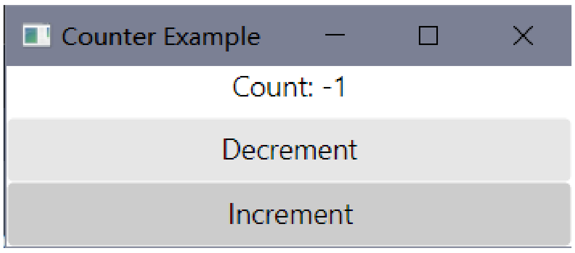


Figure 4.1: Counter App

As introduced in Section 2.1.3, the basic view function in Fable.Elmish (used in Issie) takes the model state of the application and dispatch functions as props, and then returns a ReactElement:

```
let view model dispatch: ReactElement =
    div [
        button [
            OnClick (fun _ -> dispatch Increment)
            ...
        ]
    ]
```

The sample code of FuncUI below shows a similar overall structure, only the function returns IView (XAML view component) instead of ReactElement (HTML-based)

```
let view model dispatch: IView =
    StackPanel.create [
        Button.create [
            Button.onClick (fun _ -> dispatch Increment)
        ]
        ...
    ]
```

And the sample from Fabulous.Avalonia is similar, a difference would be that the update function (dispatch) is not passed as a parameter. This abstraction, while making the code neater, could reduce clarity for those used to seeing how actions are passed up to the update function via a dispatch function. It might make it harder to trace the flow of data and control, especially in complex applications.

```
let view model: Fabulous.Avalonia.View =
    VStack() {
        Button("Increment", Increment)
        ...
    }
```

Performance:

Though both libraries implement a FRP style DSL for Avalonia, Avalonia.FuncUI implements a patching mechanism to optimize the performance [45].

FuncUI views don't hold a direct reference to their backing Avalonia controls. The view structure determines the association between a view and its control. If the structure of the view changes (e.g., a different type of control is now represented in the view hierarchy), the backing control might also change. FuncUI ensures that the new backing control is appropriately patched with all the attributes set as specified in the view, maintaining the consistency and integrity of the UI.

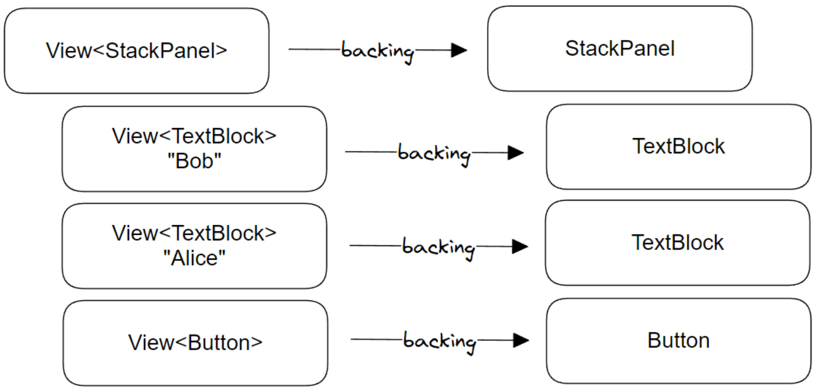


Figure 4.2: Backing Control and Update. Image credits[45]

If the type of the view in the new structure does not match the type of the backing control, FuncUI creates a new backing control to replace the old one. The new control is then initialized and patched with the appropriate attributes as defined in the view.

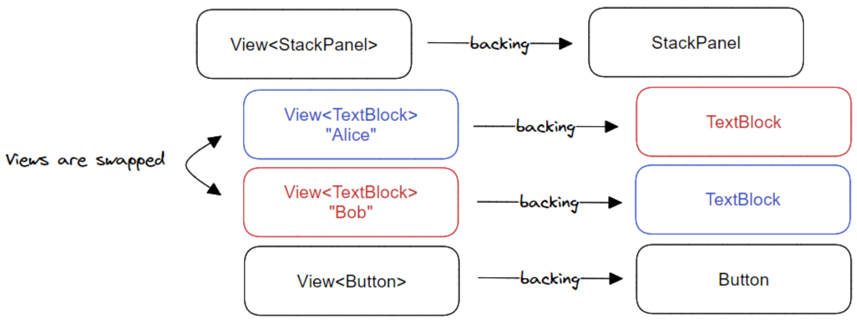


Figure 4.3: Backing Control Replacement. Image credits[45]

The life cycle of a FuncUI view excels due to its seamless integration with Avalonia controls, streamlined updates via patching mechanisms, and adaptability in handling view structures. This configuration facilitates dynamic and responsive user interfaces that can effectively mirror application state changes.

In comparison, while Fabulous offers guidance and APIs aimed at enhancing view binding and update efficiency [46], it falls short in terms of integration ease and implementation simplicity, primarily due to its insufficient documentation. These limitations make it a less viable option for projects requiring straightforward and well-documented tools.

Given FuncUI's better community support superior documentation and the aforementioned technical advantages, Avalonia.FuncUI emerges as the superior choice for our GUI wrapper. Its comprehensive features align well with our need for an efficient, maintainable, and responsive UI framework, making it the optimal selection for implementing high-performance user interfaces.

4.2 Tech-stack Comparison

The final technology combination for the .NET version of Issie would be Avalonia and Avalonia.FuncUI, we can now have an overall review of this tech stack and how the application will be built and run.

4.2.1 Application Life Time

The image 4.4 below depicts the architecture of AvaloniaUI[47]. The core library, AvaloniaUI, provides fundamental UI components and services like bindings, logical/visual tree structures, renderers, controls, and styles, based on .NET Standard 1.1 for wide compatibility.

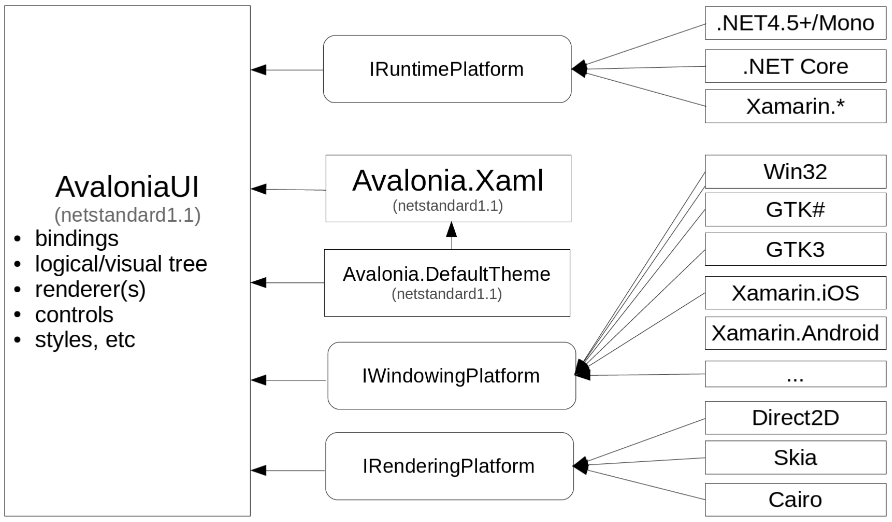


Figure 4.4: Avalonia Application Architecture. Image credits:[48].

Avalonia.XAML is the part of the framework that handles XAML processing, enabling developers to define UIs in a declarative manner, while Avalonia.DefaultTheme supplies the standard styles and templates for the UI controls.

The architecture abstracts underlying platform specifics through interfaces like IRuntimePlatform, IWindowingPlatform, and IRenderingPlatform, allowing the framework to run on different operating systems and utilize various graphics engines. Implementations of these interfaces adapt AvaloniaUI to work with .NET/Mono, .NET Core, Xamarin, and different windowing and rendering systems like Win32, GTK#, Direct2D, Skia, and Cairo, enabling a unified development experience across desktop and mobile platforms.

In contrast to Avalonia's single-process approach, Electron utilizes a multi-process architecture, as described in Section 2.1.1. This structure offers several advantages, such as improved security and isolation between components, but introduces complexities due to the need for Inter-process Communication (IPC), which can complicate adherence to the MVU paradigm.

An example of this complexity is demonstrated in how Issie handles a right-click to trigger a context menu, shown in Image 4.5 below:

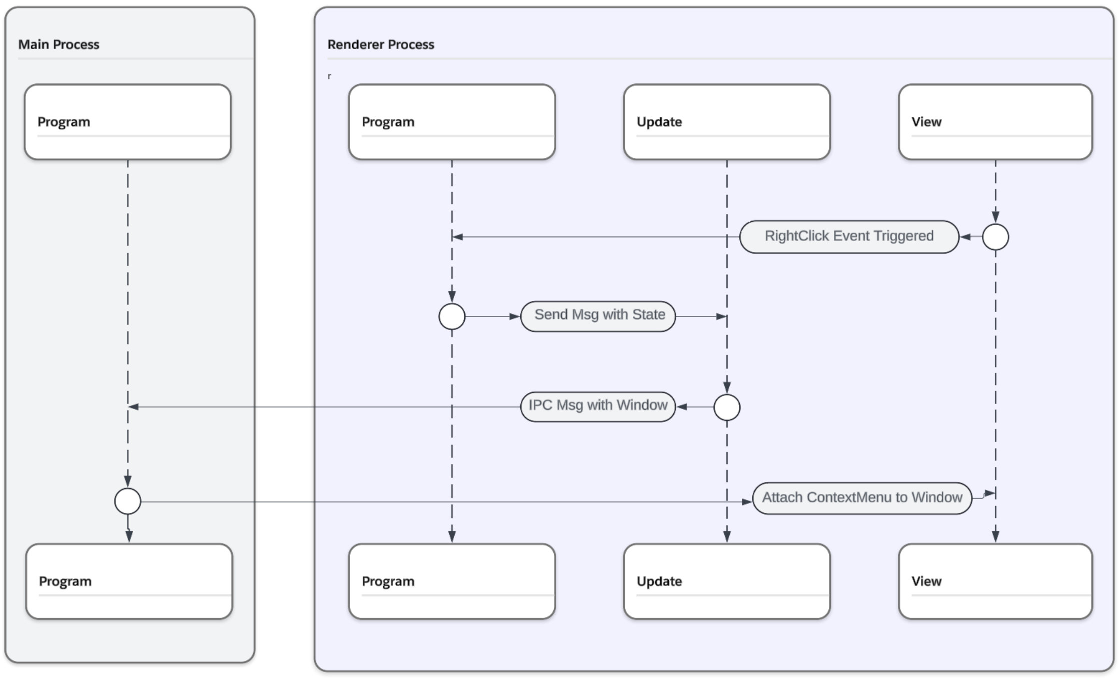


Figure 4.5: Issie Context Menu Rendering Process

This process involves an IPC message sent from the Update Module to the Main process and then directly attached to the window, bypassing the model update. Such operations indicate the challenges posed by Electron's multi-process architecture in maintaining a smooth MVU flow.

Conversely, Avalonia's architecture, which opts for a single-process design, aims to ensure uniformity across platforms and simplify development. This approach maintains a consistent single-threaded model, enhancing cross-platform consistency and reducing the complexities associated with IPC. This streamlined architecture facilitates a more MVU-compliant approach to UI updates and interactions.

If we were to reimplement the same context menu feature using Avalonia.FuncUI, the execution logic would be as follows:

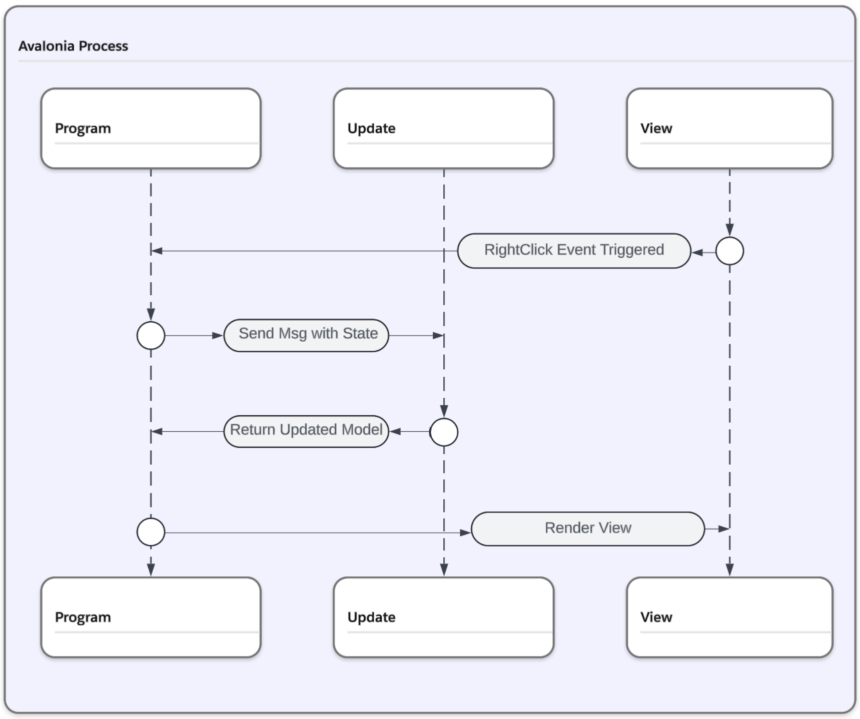


Figure 4.6: Avalonia Context Menu Rendering Process

This Avalonia implementation provides a clearer and more logically sequential process that fits better within the MVU paradigm and simplifies code complexity. Similar benefits emerge when streamlining the .NET toolchain, which we will explore in the next section 4.2.2, demonstrating further advantages in complex application scenarios.

4.2.2 Dependency Management

The nature of the fable compiler brought up difficulties in library management, two package management tools are used NuGet and Node Package Management (NPM) which produce separate dependency lock files (package.json and paket.dependencies)

Package	Description
NuGet Packages	
Fable.Browser.Css	Fable bindings for CSS.
Fable.Electron	Fable bindings for Electron APIs.
Fable.Elmish	Core library for Elmish, supporting Elmish.Debugger, Elmish.HMR, and Elmish.React.
Fable.Import.Node	Fable bindings for Node.js.
Fable.Browser.Dom	Fable bindings for DOM manipulation.
Fable.SimpleJson	Simplifies JSON serialization/deserialization.
Fable.Promise	Provides F# async programming with JavaScript promises.
Fable.React	Fable bindings for React.
FSharp.Core	Core library for F#.
Fulma	Fable bindings for the Bulma CSS framework
Thoth.Json & Thoth.Json.Net	JSON encoder/decoder focused on performance and .NET interoperability.
...	...
npm Packages	
@electron/remote	Provides remote procedure calls in Electron.
bulma	Modern CSS framework based on Flexbox.
electron	Core framework for building cross-platform desktop apps.
electron-builder	Packages and builds Electron apps.
react & react-dom	Building user interfaces and handling DOM interactions.
react-router-dom	DOM bindings for React Router.
webpack	A module bundler for JavaScript.
babel-loader	Integrates Babel transpiler with webpack.
...	...

This design also makes it hard to add new packages that we need to consider the compatibilities with fable and JS when involving interaction of DOM. The Avalonia project would have much simpler dependencies for similar implementation.

Package	**Description**
Avalonia	A cross-platform XAML-based UI framework.
Avalonia.Desktop	Extensions for desktop-specific capabilities in Avalonia.
Avalonia.FuncUI	Functional-reactive UI framework for F# with Avalonia.
Avalonia.FuncUI.Elmish	Elmish integration for state management in Avalonia.FuncUI.
Avalonia.Themes.Fluent	Fluent Design System themes for Avalonia.
Avalonia.Fonts.Inter	The Inter font family for Avalonia UIs.
Avalonia.Diagnostics	Debugging tools for Avalonia development.
Elmish	Model-View-Update architecture for F# applications.
Thoth.Json	JSON serialization tailored for F# simplicity.
Thoth.Json.Net	Thoth.Json integration with .NET's Newtonsoft.Json.

4.3 Conclusion

In this section, we have detailed our selection of Avalonia and Avalonia.FuncUI from various .NET technologies, and discussed the improvements that transitioning to a .NET implementation could bring to Issie with examples. These improvements are primarily in coding paradigms and project management. Building on the results of this analysis, we will commence the implementation process in Chapter 5. The subsequent Chapter 6 and 7 will provide a more detailed comparison focused on the performance aspects of these technological choices.

5 Implementation

Contents

Building upon the technology choices outlined in Chapter 4, the re-implementation of Issie will commence using Avalonia and Avalonia.FuncUI. Initially, we will develop a proof of concept application that addresses the primary requirements specified in Chapter 3. This stage will serve as a foundation for discussing the detailed implementation process, highlighting challenges encountered, following the assessment of the completeness of the porting effort and guidance for future development initiatives.

5.1 Proof of Concept

We developed a demo application to validate that our selection of tech stack can fulfill the technological requirements outlined in Section 3.1: SVG rendering, multi-platform compatibility, and adherence to the MVU paradigm.

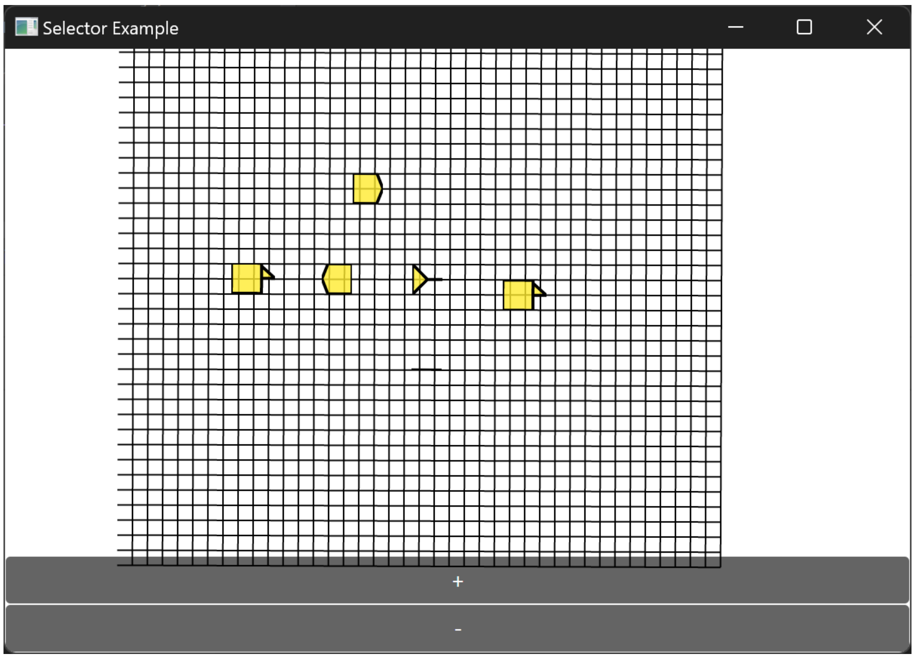


Figure 5.1: Demo App on Windows Platform

SVG Component and Transform

We explored rendering electronic symbols as SVG components within the demo app, leveraging FuncUI DSL. This approach, akin to HTML SVG polygon handling, focused on symbol generation and UI transformations, including text blocks, lines, and polygons, to ensure feature parity with Issie's existing capabilities (see Figure 5.1).

Multi-platform Compatibility

Being built to run across multiple platforms using the .NET framework, the demo verified the anticipated cross-platform compatibility. This is crucial for maintaining Issie's usability across different operating environments.

MVU Compatibility

The implementation adhered strictly to the standard Elmish MVU paradigm, confirming that our chosen tech stack is well-suited for our architectural goals.

Along with the analysis made in the previous Chapter 4, we can say that the technology requirement of the project has been validated at this stage, and meanwhile the possibility of porting *Drawblock* module at an early stage

Table 5.1: Requirements Validation

Technology Assessment	
Feature	Status
SVG rendering performance	✓
Multi-platform compatibility	✓
Tech stack compatibility	✓
Developer ecosystem	✓
Functionality Block	
Module	Status
DrawBlock	✓
Simulation	Pending
FileIO	Pending
General UI	Pending

This section confirms the viability of the selected technologies, laying a robust foundation for the full implementation phase.

5.2 Implementation Challenges

This section outlines the primary challenges encountered during the porting. The most significant challenge comes from modules directly interacting with the user interface, requiring intricate conversions between different DSL, and another unexpected challenge comes from the type system difference due to the fable trans-compilation. Additionally, compatibility adjustments are necessary for the remaining parts due to lower-level API differences.

5.2.1 DSL Differences

Many UI components like buttons, input boxes, and tabs require adjustments in property specifications. For example, the size property in Avalonia.FuncUI accepts integers (e.g., *12*) instead of strings (e.g., "*12px*") used in Fable.Elmish. These properties are often deeply integrated into the codebase and are also serialized in the schematic project files.

To facilitate the porting process and ensure backward compatibility, we introduced the AvaloniaHelpers.fs module. This module contains functions to bridge DSL differences, such as converting

SVG component specifications from strings to a list of Point data types. In section 5.4 the guidance for further development covers more detail of how to convert DSL in a standardized manner.

Still, there are bigger differences in tech stack and DSL which we can't replace directly, as we mentioned in the Analysis section with context menu example4.5, when there is an interaction with the main process of the electron, a full re-implementation is inevitable.

In Issie key and mouse event binding are made through electron (web) API like document.onkeydown.

```fsharp
let displaySvgWithZoom model dispatch ...=
    document.onkeydown <- (fun key ->
        dispatch <| (ManualKeyDown key.key)
    ...
```

This is not a standard way of handling hardware subscriptions in MVU principal, and it is also impossible to implement directly in Avalonia.FuncUI, instead of a more standard practice used here

```fsharp
let subscriptions _ =
    let keyDownSub (dispatch: Msg -> unit) =
        this.KeyDown.Subscribe(fun eventArgs ->
                dispatch (Sheet (SheetT.Msg.ManualKeyDown (eventArgs
                    .Key.ToString())))
            )
    [[ nameof keyDownSub ], keyDownSub ]

Program.mkProgram init update view'
|>  Program.withHost this
|>  Program.withSubscription subscriptions
```

5.2.2 Type System

The type system encountered several unexpected challenges during the porting process, as illustrated by the following code snippet from Issie. This code block, which generates a label for repeated component names using the Regex API, consistently triggers a type exception error in line 3 when ported to Avalonia.

```
1 let index = Regex.Match(str, @"\d+$")
2     match index with
3     | null -> 0
4     | _ -> int index.Value
```

Upon reviewing the .NET documentation on Regex, it became clear that Regex.Match does not return a null value but rather a *Match* type object. The original code functioned in Issie due to Fable compiler's type reflection, which does not strictly enforce type safety, unlike the standard .NET framework.

Another example involves the *pending* attribute of the application model, which is a list of *Msg* pending execution. This attribute is routinely compared during each MVU execution cycle. However, it includes functions and union types that fail .NET's equality checks, making them incompatible without Fable's transcompilation.

Similar issues arose in other cases where type-sensitive operations were required, such as using the *unbox* API. Solutions to these type incompatibilities varied, but generally involved adding type constraints and making appropriate conversions.

These challenges underscore the potential benefits of re-implementing in Avalonia, particularly in terms of maintainability and adherence to the standard practices of the MVU paradigm and F# programming. This approach not only resolves type safety issues but also aligns with more conventional coding practices in .NET.

5.2.3 File Management

Adapting file management functionalities to work with Avalonia involves several changes, particularly in how JSON data is handled:

- **API Replacement:** The Electron API used for reading and writing circuit schematic files (DMG) is replaced with an equivalent Avalonia API, which is relatively straightforward.
- **JSON Parsing:** The transition from Fable.SimpleJSON to Throth.JSON necessitated the development of custom encoders and decoders. These custom solutions ensure that legacy

schematic files remain compatible with the new system by accommodating differences in how map and list data types are handled.

5.3 Porting Results

After addressing the issues outlined earlier, we systematically transitioned from Fable.Elmish to Avalonia. The re-implementation of Issie on Avalonia now features a user-friendly interface and supports basic functionalities such as circuit modification and simulation, as illustrated in Figure 5.2.

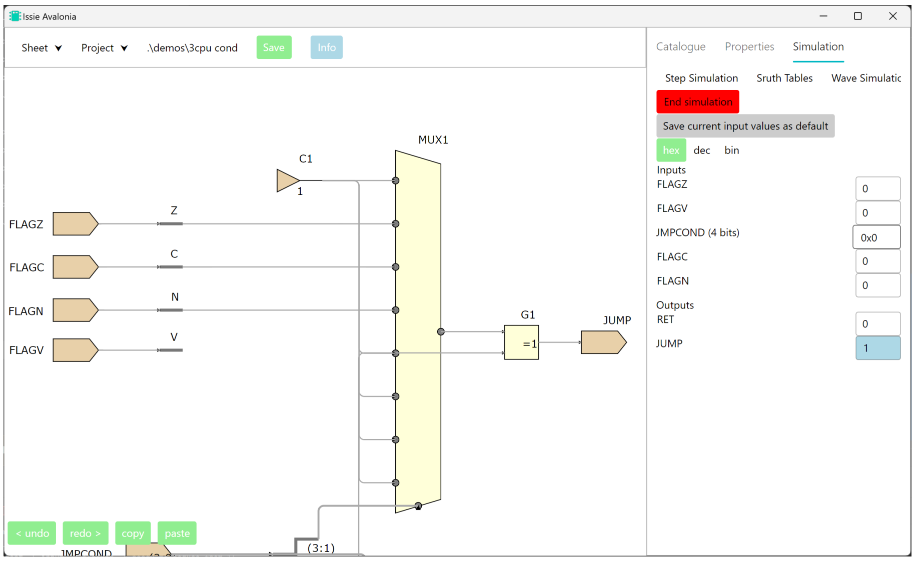


Figure 5.2: Issie Avalonia

In terms of project management and future development, we continued using the package management tool *Paket* and retained the original directory structure and filenames to facilitate easy identification of corresponding components. Currently, of the 42k lines of code across 90 major F# files, 62 files have been successfully ported to Avalonia, representing 22k lines of operational code.

Unfinished parts, as listed in Table 5.2, primarily include two other simulation modes and some interactive functions for circuits and canvas. Detailed guidance for addressing these remaining areas and future development will be discussed in the following section (Section 5.4).

Table 5.2: Functionality Completion Summary

Module	**Block**	**Functionality**	**Category**	**Complete**
Drawblock	**Sheet**	Render circuits	Rendering	✓
		Zoom canvas	Interactive	✓
		Drag canvas	Interactive	Pending
	Wire	Render wire with different widths and labels	Rendering	✓
		Automatic routing	Arithmetic	✓
		Connect/disconnect from port	Interactive	Pending
	Symbol	Render all Issie symbol types	Rendering	✓
		Move symbols	Interactive	✓
		Zoom and rotate symbols	Interactive	Pending
		Copy symbols	Interactive	✓
		Multiple symbol selection	Interactive	✓
		Overlap detection	Arithmetic	✓
Simulation	**Simulator**	Read in circuit	SystemIO	✓
		Step/Waveform output	Arithmetic	✓
	Step Simulation	Basic display of step, input, output	Rendering	✓
		Input/Step Modification	Interactive	✓
		Save/Load Simulation Config	SystemIO	Pending
	Waveform Simulation	Waveform Render	Rendering	Pending
		Wave Selection	Interactive	Pending
		RAM selection	Interactive	Pending
FileIO	**Project**	Read in demo project	SystemIO	✓
		Read in legacy project	SystemIO	Pending

Continued on next page

Module	**Block**	**Functionality**	**Category**	**Complete**
		Save project	SystemIO	✓
	Sheet	Import Sheet	SystemIO	Pending
		Copy/Paste Sheet	SystemIO	Pending
General Interface	**MainView**	Top Menu with project management functionalities	Interactive	✓
		Catalogue menu for symbol creation	Interactive	Pending
		Simulation menu	Interactive	✓
	PopUP	Project Popup window when app start	Interactive	✓
		Notifications	Interactive	Pending
		Simulation Related Popup	Interactive	Pending

5.4 Further Developments

Referring to the previous section 5.3 the structure of code and file are maintained the same at my best, for the good of the previous developer of Issie or the new developer wants a reference from the original Issie.

Meanwhile, different from the list of functionality 5.2 there is a list of files A.1 that have been or have not been ported which is put in the appendix and project documentation for developer reference.

The process of porting a new module would be: Identify a module or function in the list of functionality 5.2, which refers to the list of files that contains direct or indirect interaction of module, for existing modules there should be a *TODO* label which contains the identification of the module it belongs.

A more specific example of porting the wave simulation module is shown below:

1, Find files that directly interact with this module in the file appendix **??**, in this case, would be *WaveSim.fs* *WaveSimHelper.fs* and *WaveSimSelect.fs* which is all in the pending state which means no file created at this point.

2, For existing upper-level files like *MainView.fs* check the TODO hint left in the module, each TODO is labeled with its related module like *[WAVESIM]*.

```
let viewSimSubTab canvasState model dispatch =
    match model.SimSubTabVisible with
    | StepSim ->
        ...
    | WaveSim ->
        StackPanel.create [
            ...
            // TODO [WAVESIM] The truth table simulation view needs
                to be implemented
            // TruthTableView.viewTruthTable canvasState model
                dispatch
        ]
        ...
```

3, for new files try copy/paste from the original Issie project can be a way to start with, IDE will identify incompatible parts of code, which are usually missing attributes from other unfinished modules or DSL differences.

```
let displayUInt32OnWave wsModel (width: int) (waveValues: uint32
    array) (transitions: NonBinaryTransition array) : ReactElement
    list =
     ...
```

Which *ReactElement* will throw an undefined error because it's part of Fable.ELmish DSl and *NonBinaryTransition* are undefined because they should be imported from another unfinished module.

4, For undefined attributes, try to check the import first, then try a global search in both Issie and Issie Avalonia for the part missing. In this case you can find *NonBinaryTransition* is a method in *WaveSimHelper.fs*, the option can be either skip or comment out this part or port *WaveSimHelper.fs* at meantime.

5, For DSL difference, a guidance is in project README which includes the basic difference in DSL, and check Avalonia.FuncUI document basic component declaration, for more sophisticated interaction try checking the Avalonia API document which would solve the problem most time.

5.5 Summary

In this chapter, we confirmed that the demo application satisfies the extensive technological requirements necessary for large-scale porting. We also identified several challenges encountered during this process and provided guidance for future developers. These insights offer valuable guidance not only for the continued development of Issie on Avalonia but also for other projects utilizing similar technology stacks.

Having implemented a broad range of functionalities, we are now prepared for systematic testing and evaluation of the two technology stacks. The forthcoming chapter will detail our methodology for this assessment, as outlined in Chapter 6.

6 Test

Contents

With a wide array of functionalities now in place, we are equipped to proceed with the systematic testing and evaluation of the two technology stacks, this section mainly covers the methodology used in the testing process and how to get measurements and data for the final analysis and evaluation.

6.1 GUI Performance Test

GUI performance is a critical deliverable of this project. We will begin by establishing benchmarks for GUI testing, followed by a discussion on the tools required for these tests, ranging from a self-developed tracing module to advanced developer tools available in both Web and .NET environments for a comprehensive performance comparison.

During a circuit symbol drag interaction, there will be multiple frames rendered, leveraging developer tools from both Electron and Avalonia, as shown in image 6.1, we can accurately gauge the rendering layer performance of the GUI. A key metric in these tests is the application's frame rate, especially during interaction actions like dragging, which directly reflects the smoothness and responsiveness.

Table 6.1: Manual Render Test Methodology

	Description
Sample Selection	Uses the EEE1labs test suite, including the full implementation of the EEP1 CPU and 23 digital logic circuit designs (e.g., ALU, register file, decoder).
Sample Size	Each test case is executed 10 times to minimize variability.
Test Action	Evaluates GUI performance through user interactions such as dragging, copying, and selecting multiple components.
Measurement	Measures the time for each action using a tracing module, recording both the execution times of update and view functions and the overall frame rate using developer tools.

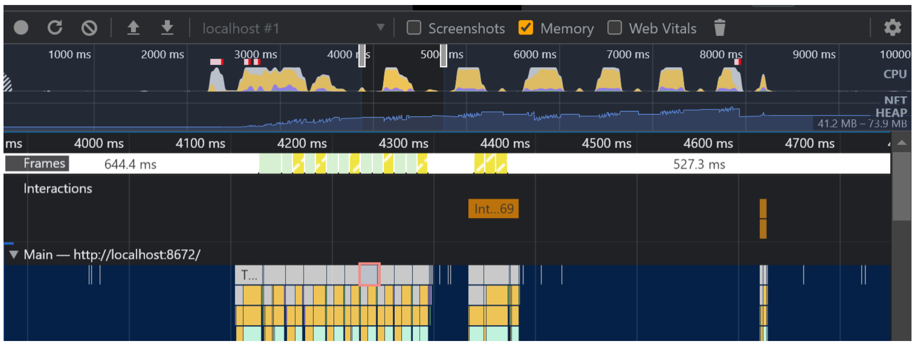
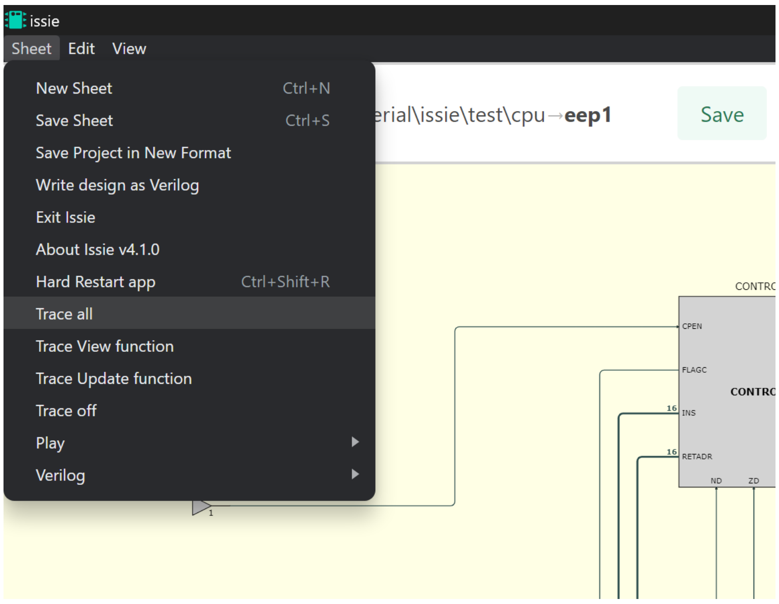


Figure 6.1: Eletron Developer Tool Frame Record

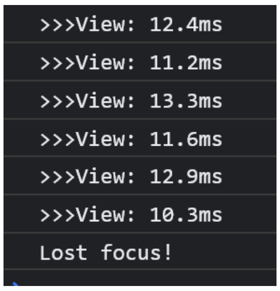
Frame Time: Providing the time each frame takes during interactions with the circuit. Combined with other information from the developer tools, we can obtain a detailed breakdown of each frame, including the view and update functions, GUI rendering, and virtual DOM updates.

Update and View Function Time: With the time helper and tracing module now integrated into the Avalonia re-implementation, we can effectively monitor and compare the performance of various application modules including model update or view function execution, which are two of the most crucial components of MVU application performance. Image 6.2a and 6.2b below demonstrates how tracing is set up in Issie's developer mode and the corresponding output.

Rendering Time: Beyond the execution time of the view function, additional time is required for properties and layout rendering by the User Interface (UI) framework. This value is usually monitored by developer tools frame by frame, as shown in Figure 6.3 below is how the Avalonia developer tool captures rendering time.



(a) Tracing Options



```
>>>View: 12.4ms
>>>View: 11.2ms
>>>View: 13.3ms
>>>View: 11.6ms
>>>View: 12.9ms
>>>View: 10.3ms
Lost focus!
```

(b) Tracing Output

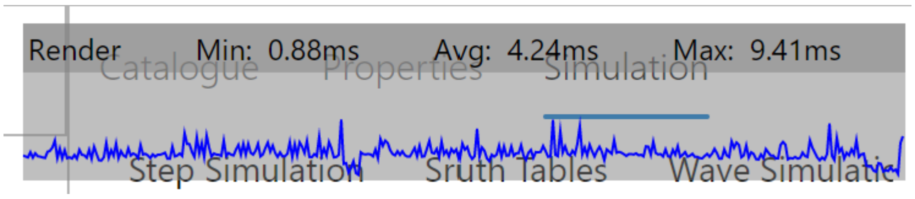


Figure 6.3: Render Time Monitoring

6.2 Simulator Performance Test

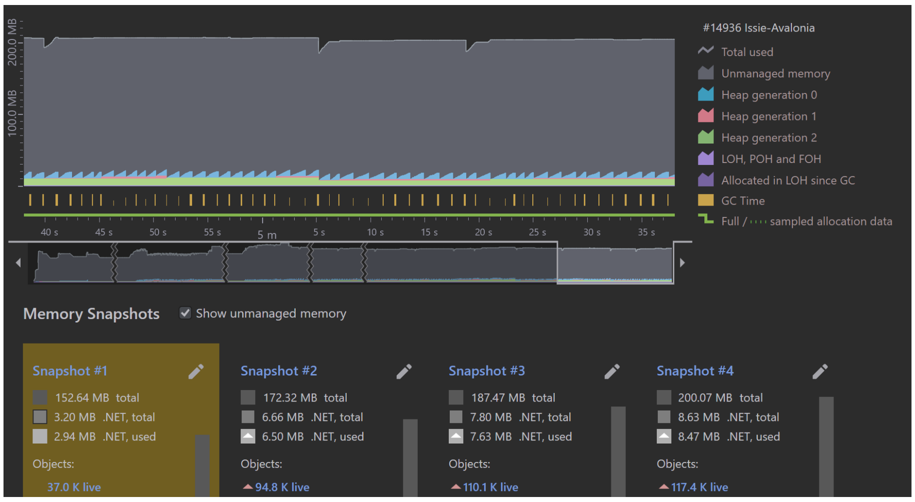
Based on the benchmark and testing module for the simulator implemented by Yujie [49], this benchmark is designed to compare the simulation speed of the new simulators with the Baseline simulator, measured in tick × component/second. All designs in EEE1labs are used as the test suite. This benchmark runs each design for 1000 clock ticks and repeats 10 times, the final score is the geometric mean of the simulation speed of each design sheet

6.3 Memory Consumption

With the developer tool as shown in image 6.4, we could have a snapshot of application memory usage at a specific point of execution, which enables us to record the memory consumption of the heap and overall usage of the app in rendering and simulation test process above.

Table 6.2: Simulation Benchmarking Methodology

	Description
Realistic Workloads	Utilizes EEE1labs test suite for comprehensive testing, including the EEP1 CPU and 23 digital logic circuit designs (arithmetic logic unit, register file, decoder, etc.).
Sample Size	Each test case is run 10 times to minimize random variations, such as those caused by Just in Time (JIT) compilation.
Test Action	Warm-up runs are executed before benchmarking to account for JIT optimizations. The geometric mean calculates speed scores due to significant variances in simulation speeds among different design sheets.



Snapshot #1
152.64 MB total
3.20 MB .NET, total
2.94 MB .NET, used
Objects:
37.0 K live

Snapshot #2
172.32 MB total
6.66 MB .NET, total
6.50 MB .NET, used
Objects:
94.8 K live

Snapshot #3
187.47 MB total
7.80 MB .NET, total
7.63 MB .NET, used
Objects:
110.1 K live

Snapshot #4
200.07 MB total
8.63 MB .NET, total
8.47 MB .NET, used
Objects:
117.4 K live

Figure 6.4: Memory Snapshot

6.4 Summary

In this chapter, we have gone through the testbench and testing tool for GUI rendering, simulator and memory performance, with these tools we can finally evaluate the overall performance of project in the following Chapter 7.

7 Evaluation

Contents

With all functionality requirements fulfilled, the benchmark established and the testing tool decided, we can now make a final evaluation of the project. The evaluation phase aims to validate the new implementation's effectiveness and highlight possible improvements over the original Issie application, evaluate the completeness of current porting, and difficulties of further works, and conclude the necessity of completing a full port to Avalonia.

7.1 Performance

7.1.1 Rendering Performance

Utilizing the methodology detailed in Section 6.1, we began by comparing the frame rates between the Electron and Avalonia applications, as summarized in Table 7.1:

Table 7.1: Frame Rate Comparison for Electron and Avalonia

Operation	Electron Frame Rate (fps)	Avalonia Frame Rate (fps)
Single Component Drag	33.12	20.01
Single Component Copy	32.31	22.83
Multi Component Drag	49.12	17.82
Multi Component Copy	51.87	18.98

To provide further insight, we converted these frame rates into the time taken for each frame, illustrating the duration required for the application to complete an MVU structure update and rendering cycle, detailed in Table 7.2:

Table 7.2: Frame Time Comparison for Electron and Avalonia

Operation	Electron Frame Time (ms)	Avalonia Frame Time (ms)
Single Component Drag	18.17	30.19
Single Component Copy	18.48	30.95
Multi Component Drag	20.36	56.12
Multi Component Copy	19.29	52.69

This data demonstrates that Electron's Issie implementation offers superior responsiveness during component interactions, aligning more closely with user perceptions of component movement. However, not all sheets perform at similar speeds. Some sheets in the EEP1 sample set showed better performance, hinting at underlying factors affecting this discrepancy. Consequently, we conducted additional tests varying the number of displayed symbols and size of loaded projects to explore the correlation between performance.

The test below is designed to explore the relationship between frame time and loaded project size. Figure 7.1 depicts the comparative performance of both applications on a smaller scale, displaying a simple half-adder sheet across different project sizes:

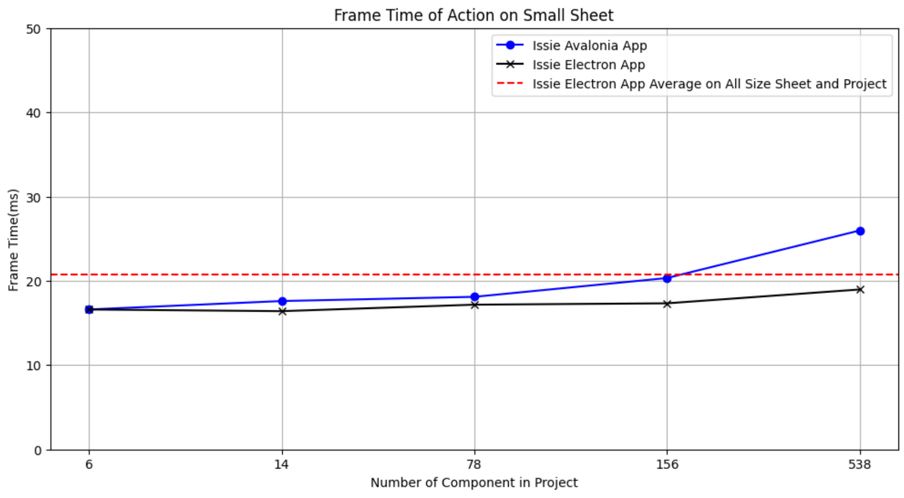


Figure 7.1: Performance Comparison on Different Project Sizes

In the scenario of a small circuit displayed, the performance of the two Issie versions is comparable. However, in scenarios such as the final test case with 538 components, Avalonia's application shows slower speeds, highlighting the impact of project size on rendering performance, a possible reason is that even though most of the symbols are not rendered, the larger size of the model would reduce the efficiency of model update.

As shown in Figure 7.2, increasing the size of the circuit displayed on the sheet leads to a notable increase in frame time. This degradation in performance in larger sheets could explain the results observed in Table 7.2, given that our test circuit, EEP1, primarily consists of large circuit sheets.

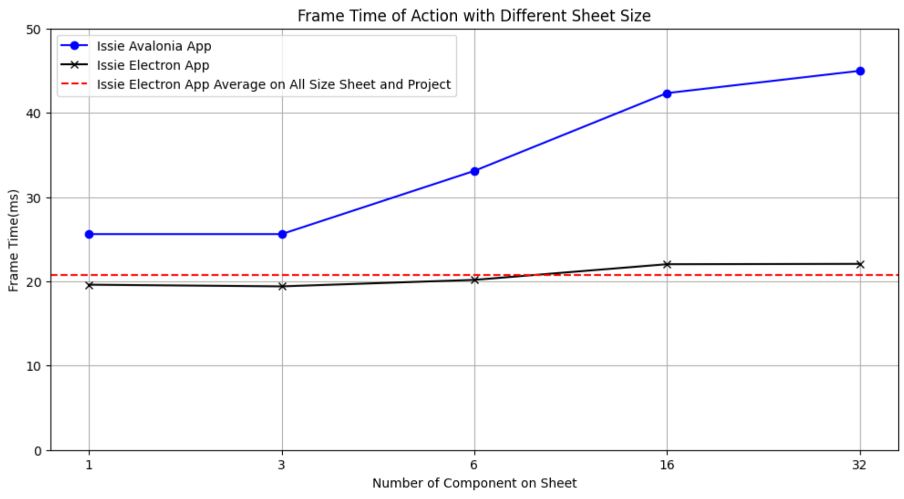


Figure 7.2: Performance Comparison on Different Sheet Display

When the complexity of the interaction is increased by adding more components and wires, as illustrated in Figure 7.3, there is again a rise in frame time, providing a clearer demonstration of the performance issues with multiple component interactions documented in Table 7.2.

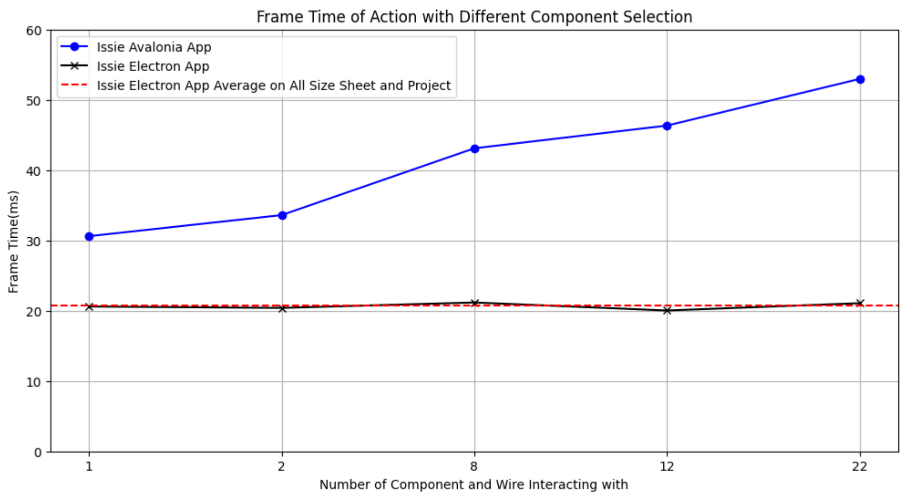


Figure 7.3: Performance Comparison on Different Component Interacting

Further Analysis

Based on the observations above, we can conclude that the number of symbols displayed and the number of symbols interacted with have a major impact on GUI performance in terms of frame rate, while the overall size of the project has a smaller impact on performance.

To further understand these issues, we conducted a detailed breakdown of each result in Figure 7.3, by collecting data on the execution times for view and update functions, rendering, and remaining script execution time in each frame as previously described in Section 6.1. In Figure 7.4, it is evident that while the number of components interacted with increases, the view function's execution time remains stable and close to the Electron app's performance. However, the rendering time, although stable, is higher than the Electron app, and there is a significant increase in the time required for the update function and other script execution.

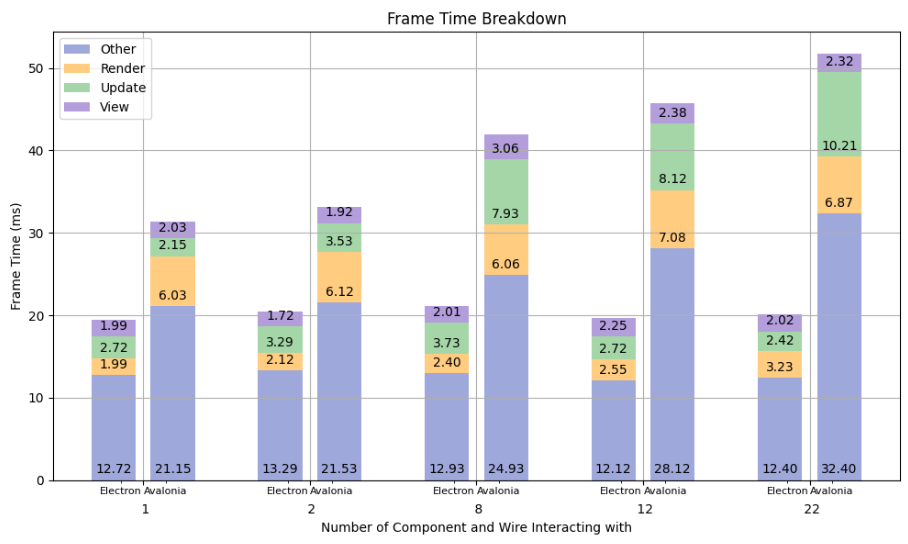


Figure 7.4: Performance Comparison with Time Breakdown

As we can see, the update and other script execution times are crucial in the current case. Two main aspects can potentially be optimized: the update function and component caching used in Issie.

Starting with the update function, the re-implementation of Issie on Avalonia primarily modifies the *View* part of the original MVU structure. Logical operations in the *Update* function generally remain the same. However, while the new *View* function's execution time is close to the Electron app's, the *Update* function time accumulates during the test. This behavior is potentially related to event handling and the message queue of the Avalonia application and will be a critical area for future improvement.

Regarding component caching, as introduced in Section 2.1.3, the Fable.Elmish app uses React Virtual DOM and Functional Components, which significantly improve rendering time. In FuncUI.Avalonia, we use a Patching Mechanism (see Section 4.2.1), similar to the concept of Virtual DOM. For the Function Component, which caches all symbols and wires during the update, we used FuncUI Component [50] that shares the same idea. However, based on the test results above, this may not be a mature implementation, indicating a key area for further optimization.

Summary

Currently, the Avalonia implementation of Issie delivers comparable performance in rendering and interacting with simple circuits. However, in more complex scenarios, achieving a frame rate of around 20fps does not meet our objectives for optimizing Issie's performance on the new platform. Based on further analysis and test results, we have identified several potential solutions. Addressing these performance issues will likely be a central focus for future development and optimization efforts.

7.1.2 Simulation

As outlined in Section 6.2, simulation performance is evaluated in terms of Ticks*Components per second, effectively balancing the considerations of time and complexity in the simulation process. This measure provides a stark contrast to the rendering performance discussed in Section 7.1.1, where significant improvements have been observed.

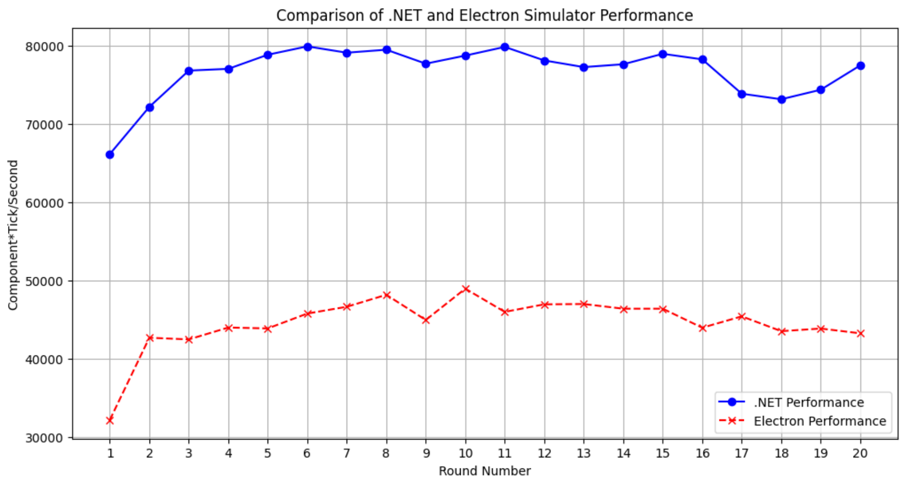


Figure 7.5: Performance Comparison on Simulator

As depicted in Figure 7.5, the Avalonia simulator reaches a peak performance of 79,871, significantly surpassing the Electron simulator, which achieves only 61.23% of Avalonia's output. Notably, both platforms demonstrate a gradual increase in simulation speed initially. This progressive enhancement is primarily due to the Just-In-Time (JIT) compiler, which requires a warm-up period to fully optimize execution speeds. This critical optimization process is instrumental in delivering the high performance observed in more complex simulation scenarios.

7.1.3 Memory

In terms of memory performance, the ROM space required by the application has not significantly changed with the transition to Avalonia. However, notable improvements have been observed in

RAM usage, which are clearly detailed in Table 7.3. We see a 22.3% reduction in the overall RAM consumption of the application, which includes substantial decreases in both the Simulator and Managed Heap sizes, reflecting more efficient memory management.

Table 7.3: Memory Performance in MB (Mega Bytes)

Stage	Issie			Issie Avalonia		
	Simulator	Heap	Overall	Simulator	Heap	Overall
On start	0.8	16.3	190.5	0.6	16.2	152.4
Project Load	1.1	19.8	259.9	0.8	15.8	207.9
Start simulation	4.3	20.1	240.2	3.4	16.0	192.1
10 Ticks	5.9	19.9	239.0	4.2	15.9	191.2
100 Ticks	6.2	20.4	231.0	4.6	16.3	184.0
1000 Ticks	5.7	19.6	249.7	4.5	15.6	199.7

7.2 Implementation Evaluation

Having obtained the full test results and completed our analysis, we now review the entire porting process, including the pros and cons identified, the challenges encountered, and considerations for future work.

7.2.1 Tech Stack and Project Management

As anticipated during the initial analysis phase in Section 4.1.1, the porting process was smooth in terms of coding style and design. Avalonia.FuncUI provided a solid and user-friendly DSL. The results listed in Section 5.3 show that the project structure remains consistent with the Issie Electron project, while also reducing overall complexity by integrating the multi-process Electron program into a single process and simplifying dependency management.

7.2.2 Implementation Challenges

In Section 5.2, we discussed the challenges faced during the porting process. The primary challenge and workload of converting the DSL have been mostly addressed, except for some functionalities that are not yet implemented. The API and type system differences still pose a problem, especially the type errors caused by the previous dependency on Fable's trans-compilation, as exemplified in Section 5.2.2. These can be difficult to source and lack a common solution for each issue. It is hoped that this report will serve as a reference for developers encountering similar issues in the future.

7.2.3 Functionality Completeness and Future Works

As indicated in Table 5.2, the fundamental functionality requirements for Issie Avalonia have largely been met, allowing for a robust evaluation of performance at this stage. However, certain critical features necessary for a fully functional Circuit CAD and Simulator Application remain absent. Notably, functionalities such as modifying wire connections and a catalogue menu for creating new symbols have yet to be implemented. Detailed guidance for addressing these gaps is provided in Section 5.4.

Further enhancements, such as the rendering optimizations discussed in Section 7.1.1, are crucial to enhancing the practical value of this re-implementation for users.

7.3 Summary

The transition to a new and simpler tech stack has notably enhanced performance in areas like simulation speed and memory usage. This change also brings several benefits, including more efficient dependency management and a smoother learning curve for F# developers, particularly for students in the Imperial HLP module.

However, GUI performance still requires significant improvement. Only after testing and implementing potential solutions outlined in Section 7.1.1 can we determine whether to continue this re-implementation on the new tech stack. Further development also faces considerable challenges, as the original project underwent extensive testing and optimization over many years. Completing the remaining functional modules is expected to be both time-consuming and resource-intensive.

Conclusions

In this project, we evaluated various .NET tech stacks and successfully re-implemented key functionalities of Issie using Avalonia. Through extensive testing in rendering, simulation, and memory usage, we have confirmed that transitioning to the .NET platform significantly enhances simulator performance and optimizes memory utilization. However, the rendering tests highlight a need for further optimization, which will be a primary focus for future developments.

Avalonia provides developers with a streamlined toolchain that simplifies dependency management and closely aligns with the .NET standard and the MVU paradigm. Despite the considerable challenges that further development may pose, the long-term benefits and potential enhancements strongly support the continuation of this port.

A

Appendix

Table A.1: File Completeness and Reference

Module	**Block**	**File**	**Completeness**
DrawBlock	Sheet	Sheet.fs	✓
		SheetSnap.fs	✓
		SheetDisplay.fs	Partial
		SheetUpdateHelpers.fs	✓
		SheetUpdate.fs	✓
	Symbol	Symbol.fs	✓
		SymbolView.fs	✓
		SymbolHelpers.fs	✓
		SymbolPortHelpers.fs	✓
		SymbolResizeHelpers.fs	✓
		SymbolReplaceHelpers.fs	✓
		SymbolUpdate.fs	✓
	Wire	BusWire.fs	✓
		BusWireUpdateHelpers.fs	✓
		BusWireRoutingHelpers.fs	✓
		BusWireRoute.fs	✓
		BusWireSeparate.fs	✓
		BusWireUpdate.fs	✓
	General	BlockHelpers.fs	✓
		PopupHelpers.fs	✓
		RotateScale.fs	✓
Common	-	EEExtensions.fs	✓
		Optics.fs	✓
		ElectronAPI.fs	✓
		HashMap.fs	✓
		CommonTypes.fs	✓
		DrawHelpers.fs	Partial
		Helpers.fs	✓
		TimeHelpers.fs	✓
		WidthInferer.fs	✓
UartFiles	-	BuildUartHelpers.fs	Pending
Verilog Component	Verilog	VerilogAST.fs	Pending
		VerilogTypes.fs	Pending

Continued on next page

Table A.1 – continued from previous page

Module	Block	File	Completeness
Simulation	Simulator	SimulatorTypes.fs	✓
		TruthTableTypes.fs	✓
		NumberHelpers.fs	✓
		SynchronousUtils.fs	✓
		Extractor.fs	✓
		CanvasStateAnalyser.fs	✓
		SimulationGraphAnalyser.fs	✓
	Fast	FastCreate.fs	✓
		FastReduce.fs	✓
		FastRun.fs	✓
		Builder.fs	✓
		Verilog.fs	✓
		Runner.fs	✓
		DependencyMerger.fs	✓
		Simulator.fs	✓
UI	General	ModelHelpers.fs	✓
		Style.fs	Partial
		Notifications.fs	Pending
		UIPopups.fs	Pending
		MemoryEditorView.fs	Pending
		MenuHelpers.fs	✓
		MiscMenuView.fs	Partial
		TopMenuView.fs	✓
		CustomCompPorts.fs	Pending
		SimulationView.fs	✓
	Truth Table	TruthTableReduce.fs	Pending
		TruthTableCreate.fs	Pending
		ConstraintReduceView.fs	Pending
		TruthTableView.fs	Pending
		TruthTableUpdate.fs	Pending
	Wave Simulation	WaveSimHelpers.fs	Pending
		WaveSimStyle.fs	Pending
		WaveSimSelect.fs	Pending

Continued on next page

Table A.1 – continued from previous page

Module	**Block**	**File**	**Completeness**
		WaveSim.fs	Pending
	View	BuildView.fs	Pending
		CatalogueView.fs	✓
		SelectedComponentView.fs	Pending
		MainView.fs	✓
		ContextMenus.fs	Partial
		UpdateHelpers.fs	✓
		Update.fs	✓

Bibliography

[1] *What is ISSIE?* `https://tomcl.github.io/issie/`, Accessed on 2024-02-01.

[2] Microsoft, *.NET Framework Documentation*, `https://docs.microsoft.com/en-us/dotnet/framework/`, Accessed on 2024-01-31.

[3] *F# Programming Language*, `https://fsharp.org/`, Accessed on 2024-01-31.

[4] S. Fowler, "Model-view-update-communicate: Session types meet the elm architecture," 14:1–14:28, 2020. DOI: `10.4230/LIPIcs.ECOOP.2020.14`.

[5] *What is Fable?* `https://fable.io/`, Accessed on 2024-01-31.

[6] *Electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS*, `https://www.electronjs.org/`, Accessed on 2024-01-31.

[7] *Elmish: Elm architecture in F#*, `https://elmish.github.io/elmish/`, Accessed on 2024-01-31.

[8] *JavaScript*, `https://developer.mozilla.org/en-US/docs/Web/JavaScript`, Accessed on 2024-01-31.

[9] Marco Selvatici, *DEflow: An Extensible Hardware Design Platform for Teaching Digital Electronics*, `https://github.com/tomcl/issie/blob/master/docsrc/files/marco-report.pdf/`, Accessed on 2024-01-31.

[10] K. Kredpattanakul and Y. Limpiyakorn, "Transforming javascript-based web application to cross-platform desktop with electron," pp. 571–579, 2018. DOI: `10.1007/978-981-13-1056-0_56`.

[11] *GNU General Public License, version 3 (GPL-3.0)*, `https://www.gnu.org/licenses/gpl-3.0.en.html`, Accessed on 2024-01-31.

[12] *Node.js Official Website*, `https://nodejs.org/en/`, Accessed on 2024-01-31.

[13] *Chromium Official Website*, `https://www.chromium.org/`, Accessed on 2024-01-31.

[14] S. Tilkov and S. Vinoski, "Node.js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, pp. 80–83, 2010. DOI: `10.1109/MIC.2010.145`.

[15] D. Alic, S. Omanovic, and V. Giedrimas, "Comparative analysis of functional and object-oriented programming," *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 667–672, 2016. DOI: `10.1109/MIPRO.2016.7522224`.

[16] D. Syme, "The early history of f#," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–58, 2020. DOI: `10.1145/3386325`.

[17] M. Learn, *Unit testing f# in .net core with dotnet test and mstest*, `https://learn.microsoft.com/en-Us/dotnet/core/testing/unit-testing-fsharp-with-mstest`, Accessed on 2024-02-05, 2021.

[18] *Fable bindings for electron apps*, `https://github.com/fable-compiler/fable-electron`, Accessed on 2024-01-31.

[19] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for guis," *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013. DOI: `10.1145/2491956.2462161`.

[20] Facebook, Inc., *ReactJS Official Website*, `https://reactjs.org/`, Accessed on 2024-01-31.

[21] *Hypertext Markup Language (HTML)*, `https://developer.mozilla.org/en-US/docs/Web/HTML`, Accessed on 2024-01-31.

[22] *Announcing Fable React 5*, `https://fable.io/blog/2019/2019-04-16-Announcing-Fable-React-5.html`, Accessed on 2024-02-05.

[23] *C# Programming Language*, `https://docs.microsoft.com/en-us/dotnet/csharp/`, Accessed on 2024-02-05.

[24] *Visual Basic Programming Language*, `https://docs.microsoft.com/en-us/dotnet/visual-basic/`, Accessed on 2024-02-05.

[25] Facebook, Inc., *Avalonia vs Electron*, `https://stackshare.io/stackups/avalonia-vs-electron`, Accessed on 2024-05-31.

[26] *QtSharp - Qt bindings for .NET*, `https://github.com/ddobrev/QtSharp`, Accessed on 2024-01-31.

[27] Microsoft, *Windows Presentation Foundation*, `https://docs.microsoft.com/en-us/dotnet/desktop/wpf/`, Accessed on 2024-01-31.

[28] *Avalonia - A cross platform XAML Framework*, `https://avaloniaui.net/`, Accessed on 2024-01-31.

[29] Microsoft, *XAML Official Documentation*, https://docs.microsoft.com/en-us/dotnet/desktop/xaml/, Accessed on 2024-01-31.

[30] *Desktop Apps with Avalonia and F#*, https://blog.tunaxor.me/blog/2020-01-23-desktop-apps-with-avalonia-and-fsharp-4n21.html, Accessed on 2024-02-05.

[31] *F# UI App with FuncUI: A Real Application*, http://blog.kevindayprogramming.com/a-real-world-fsharp-ui-app-with-funcui/, Accessed on 2024-02-05.

[32] *Fabulous Documentation*, https://docs.fabulous.dev/, Accessed on 2024-05-31.

[33] *Getting Elmish in .NET with Elmish.WPF*, Accessed on 2024-02-05. [Online]. Available: https://www.compositional-it.com/news-blog/elmish-wpf/.

[34] A. Nosenko, *WPF in 2021: Alive, Dead, or on Life Support?* Accessed on 2024-02-05. [Online]. Available: https://dev.to/noseratio/the-signs-of-wpf-currently-being-on-life-support-1h3a.

[35] *Uno Platform - Build Mobile, Web, and Desktop Apps*, https://platform.uno/, Accessed on 2024-01-31.

[36] *Avalonia vs UNO*, Accessed on 2024-02-05. [Online]. Available: https://stackshare.io/stackups/avalonia-vs-uno.

[37] Microsoft, *.NET Multi-platform App UI*, https://docs.microsoft.com/en-us/dotnet/maui/, Accessed on 2024-01-31.

[38] M. Learn, *What's new in .NET 6*, Accessed on 2024-02-05. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-6.

[39] *Fabulous for .NET MAUI (Microsoft.Maui.Controls)*, Accessed on 2024-02-05. [Online]. Available: https://github.com/fabulous-dev/Fabulous.MauiControls.

[40] Mike James, *Avalonia UI and MAUI - Something for everyone*, https://avaloniaui.net/Blog/avalonia-ui-and-maui-something-for-everyone, Accessed on 2024-05-31.

[41] Microsoft, *Blazor Official Website*, https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor, Accessed on 2024-05-31.

[42] Mike James, *A Year in Recap: Celebrating this years milestones*, https://www.avaloniaui.net/Blog/a-year-in-recap-celebrating-this-years-milestones, Accessed on 2024-05-31.

[43] *Avalonia.FuncUI - Functional Avalonia UI Framework*, https://github.com/AvaloniaCommunity/Avalonia.FuncUI, Accessed on 2024-01-31.

[44] *Introduction to Domain-Specific Languages (DSLs)*, https://en.wikipedia.org/wiki/Domain-specific_language, Accessed on 2024-02-02.

[45] *FuncUI Documentation - View Lifetime*, https://funcui.avaloniaui.net/view-basics/lifetime, Accessed on 2024-01-31.

[46] *Fabulous Documentation - Performance Optimization*, https://docs.fabulous.dev/advanced/performance-optimization, Accessed on 2024-05-31.

[47] Microsoft, *AvaloniaUI Documentation - High level architecture overview*, https://github.com/AvaloniaUI/Avalonia/wiki/High-level-architecture-overview, Accessed on 2024-01-31.

[48] *Avalonia High-Level Architecture Overview*, https://github.com/AvaloniaUI/Avalonia/wiki/High-level-architecture-overview, Accessed on 2024-02-05.

[49] Y. Wang, *A high performance digital circuit simulator for ISSIE*, Accessed on 2024-02-05. [Online]. Available: https://github.com/tomcl/issie/blob/master/docsrc/1714652_Rpt_A%20high%20performance%20digital%20circuit%20simulator%20for%20ISSIE.pdf.

[50] *FuncUI Documentation - Component*, https://funcui.avaloniaui.net/components, Accessed on 2024-01-31.