Exploring hiphop.js and Web-Based Reactive Programming

Final Report of Minor Project submitted by

Maxime Durand 2024VST9513
Under the guidance of
Professor Sanjiva Prasad

Indian Institute of Technology Delhi (Exchange Program)
IMT Atlantique
May 2025

Abstract

This project explores web programming through structured experimentation and self-guided learning. With little prior experience in web technologies, my initial goal was to develop a working understanding of front-end development and event-driven systems. On my supervisor's suggestion, I began with `hiphop.js`, a library for synchronous reactive programming. Although hiphop.js shaped the project's theoretical foundation, much of the work focused on building broader skills in HTML, CSS, JavaScript, and state management.

I developed a few suites of small applications, including hiphop-based examples demonstrating core concepts, ABRO control patterns, finite state machines, and more advanced interfaces such as a clock and an alarm app—the latter being the only one using hiphop.js to coordinate clock pulses and alarm logic.

The project also involved using various tools and libraries, including npm packages and browser dev tools, and reviewing key papers on reactive systems. While hiphop.js was not central to most implementations, working with it helped clarify core concepts in reactive programming. Overall, the experience led to a clearer, though still basic, understanding of web programming. Future work may involve integrating hiphop.js with modern frameworks or exploring more performance-conscious designs.

Contents

Chapter 1

Introduction

1.1 Motivation and Background

Web programming plays a central role in the development of interactive applications, yet it was a domain I had very little experience with prior to this project. My primary motivation was to explore and gain familiarity with the fundamentals of modern web development through a hands-on, project-driven approach. Upon the suggestion of my supervisor, I began investigating `hiphop.js`, a JavaScript library designed for reactive, synchronous programming. Although my initial focus was on learning this specific tool, it quickly became apparent that a broader understanding of core web technologies—HTML, CSS, JavaScript, state handling, and asynchronous behavior—was necessary to make meaningful progress. As a result, the project evolved to include a variety of experiments and applications, some using hiphop.js and others relying on more conventional approaches.

1.2 Problem Definition and Objectives

The core aim of this project was to bridge my lack of prior knowledge in web programming by learning through implementation. This included:

- Gaining a working understanding of fundamental web technologies and their interactions.
- Exploring the principles of reactive programming using `hiphop.js`.

- Designing and implementing a range of small programs and interactive pages, grouped by theme or technical approach.
- Investigating the role of signal-based execution and synchronous coordination in web applications.

Rather than focusing on building a single, large application, the work was divided into a series of smaller, focused projects. These allowed for clearer experimentation and learning, particularly in areas like finite state modeling, event synchronization, and interface responsiveness.

1.3 Report Layout

The structure of this report is as follows:

- **Chapter 2** introduces `hiphop.js`, including a description of its purpose, common use cases, and theoretical basis. It also defines key keywords used in the language and refers to example implementations.And link to appendix 1 that provides a detailed walk-through of hiphop.js mechanics using the ABRO examples as a guide.
- **Chapter 3** explains the project setup in detail, including the development environment, tools, and libraries used, as well as components tested but ultimately unused.
- **Chapter 4** discusses the learning process related to HTML/CSS and JavaScript modules, and includes early experiments such as page integrations and basic FSMs.
- **Chapter 5** focuses on finite state automatons: an introduction to their principles, followed by references to a suite of illustrative applications and a final, more complete FSM.
- **Chapter 6** describes the development of applications like the alarm, world clock, and chronometer, with emphasis on their use cases and interactive design.
- **Chapter 7** reflects on the knowledge gained through the project and how initial challenges were addressed.

- **Chapter 8** concludes the report with final observations and potential directions for future work.

Chapter 2

Understanding `hiphop.js`

2.1 What is `hiphop.js`?

`hiphop.js` is a synchronous reactive domain-specific language (DSL) embedded in JavaScript. It brings deterministic concurrency, instant signal propagation, and preemption to web programming. Inspired by Esterel, it allows developers to write temporal logic for coordinating event-based behavior, while while still leting the possibility to use JavaScript's native asynchronous features. `hiphop.js` is designed to simplify the orchestration of complex behaviors which are otherwise difficult to manage using conventional callback or promise-based code.[1]

The way `hiphop.js` works is that it is a model of logical time—program execution broken into a series of "instants" called "reactions". In these reactions all relevant events are processed synchronously. This should avoid pitfalls of JavaScript event-based programming. It is especially well suited for scenarios requiring deterministic parallelism and strict signal coordination.

The language has been used in real-time musical performance systems and interactive applications, as well as in smaller, structured examples like login forms or control panels [1]. It should allow both high-level orchestration and integration with low-level asynchronous behavior using constructs such as `exec` and `await`, providing a smooth blend of synchronous and asynchronous paradigms.

However, `hiphop.js` also inherits some of the complexities of its Esterel roots, especially with regard to causality errors—logical contradictions that can arise when signal dependencies form cycles. These issues are addressed by dedicated debugging support in the language's runtime [5].

2.2 Basic Constructs and Keywords

In `hiphop.js`, the main unit of communication is the *signal*, which can be either pure (present/absent) or valued (carrying data). Signals are declared in a module's interface as `in`, `out`, or `inout`. Each program is made of modules, which describe behavior over time using a synchronous model.

Key control flow keywords include:

- `emit`: Sends a signal in the current instant.
- `await`, `yield`: Wait for an event or the next instant.
- `every`: Reacts to repeated occurrences of a signal.
- `abort`, `suspend`: Used for preemption and cancellation.
- `fork ... par`: Runs code in parallel threads.
- `run`: Executes a submodule.
- `exec`: Launches asynchronous JavaScript code with lifecycle hooks. (might be outdated)

A more detailed list and descriptions of all constructs are available in the official documentation [4].

One of the central components of this project was a suite of ABRO programs developed specifically to explore and illustrate key `hiphop.js` concepts. These programs, which each consist of a HipHop module and a corresponding JavaScript driver to trigger reactions, served both as a learning tool for myself and as educational examples for others. A detailed discussion of these programs and the behaviors they demonstrate is provided in Appendix A. .

2.3 On Debugging and Causality

As explained in the paper on causality error tracing [5], one major challenge in synchronous languages like `hiphop.js` is causality errors—situations where the presence or absence of a signal cannot be determined in a logically consistent way. These can occur due to circular dependencies in parallel branches or contradictory conditionals.

An example:

```
hiphop module incoherent(inout S) {
  if (!S.now) emit S();
}
```

This program is incoherent because it emits signal `S` only if it is absent, which would make it present—a contradiction. Such programs are rejected at runtime in `hiphop.js`. Understanding and avoiding such errors required some effort and was part of the learning curve during this project.

"hiphop.js" provides runtime tools to detect and report causality cycles, which helped me correct several of my early errors.

Chapter 3

Setup and Development Environment

3.1 Project Setup and Execution Environment

The project was developed entirely in a Linux-like environment using the Windows Subsystem for Linux (WSL). This was a requirement due to the fact that the `hiphop.js` library is only supported on Linux and macOS systems. The official release of `hiphop.js` is not yet available on `npm`, so I installed it manually using the following command after setting up Node.js:

```
npm install https://www-sop.inria.fr/members/Manuel.Serrano/software/npmx/hiphop.t
```

Due to this setup, I naturally adopted the Node.js ecosystem for executing all programs. My typical workflow involved writing programs in `.mjs` modules and executing them using:

```
node main.js
```

To display web-based outputs or host static HTML/CSS/JS files for my project pages, I used the built-in `http` module in Node.js. This was done through a single `node-server.mjs` file that created a simple HTTP server to serve each webpage.

Although I briefly explored using the `express` framework, I found it less intuitive for this particular project. The abstraction layers it introduced were unnecessary for my needs, and I found debugging simpler with the more explicit and linear control flow of the built-in `http` module.

Another important aspect of the setup was the module system. The `hiphop.js` documentation clearly indicates that the library requires programs to use ECMAScript modules (ESM). As a result, I explicitly added the following line in my `package.json` file to ensure that my project used ESM:

```
"type": "module"
```

This was essential to avoid compatibility errors and allow the usage of `import` statements rather than Node's default `require()` system.

A comprehensive guide on how to set up the `hiphop.js` module along with a simple starting example are available to the following link [7]:

`http://hop.inria.fr/home/hiphop/index.html`

3.2 Exploration of Esterel and Synchronous Programming Concepts

In an effort to deepen my understanding of synchronous programming, particularly the foundations that inspired `hiphop.js`. I explored the language Esterel, which is a seminal synchronous programming language developed for reactive systems.I was oriented to the presentation *"The Esterel Synchronous Programming Language and Its Application to Real-Time Systems"* by Stephen A. Edwards [3], which gave a structured overview of key concepts like logical instants, signal broadcasting, and the causality problem.

The motivation behind this exploration was to experiment with Esterel directly by running and analyzing example programs—especially the kind that demonstrate preemption, concurrency, and causality errors—so I could better grasp the underlying semantics that `hiphop.js` attempts to embed in a JavaScript environment.

Installation Challenges

Despite having access to a working Esterel V5_92 archive, setting up the compiler proved to be significantly difficult due to a lack of updated tooling and native support for modern systems. I initially tried to run the compiler on Windows directly but encountered numerous compatibility issues, particularly due to Esterel's reliance on 32-bit binaries and specific Tcl/Tk versions.

To work around this, I transitioned to using WSL (Windows Subsystem for Linux) on Windows 11. This gave me a proper Unix-like environment more suited to Esterel's toolchain. I followed the traditional Unix installation approach outlined in the official `README.txt`, including extracting the archive, attempting to run the provided GUI installer `setup`, and eventually resolving its dependency on `wish8.2` by linking it to `wish8.6`.

Outcome

I was not able to fully simulate or interact with t. Nevertheless, this experience was valuable in exposing the complexity and limitations of using legacy synchronous systems like Esterel today.

It also offered clear insight into why newer tools like `hiphop.js`, which embed synchronous constructs in modern JavaScript environments, are useful for web-based and interactive applications.

3.3 Availability on GitHub and Docker Integration

The entire project is available online at the following GitHub repository[2]:

`https://github.com/Maxime-cod/hiphop-and-webdev-project.git`

To make it easier to run the project in consistent environments, I also created a Docker container that replicates my development setup. This allows users to build and run the project even if they are not on Linux or macOS. The Docker container uses Node.js as the base image and includes all necessary dependencies for executing the programs via the command line.

However, this setup is mainly suited for running and testing the programs—it is not ideal for reading or editing the code interactively. For that, it is preferable to clone the repository and run the project directly using Node.js inside a proper code editor.

3.4 Other Tools and Libraries Used

While the core of this project revolved around experimenting with `hiphop.js`, I also explored several supporting JavaScript libraries to enhance functionality or to better understand module integration and web-based programming.

Axios

One of the first libraries I experimented with was `axios`, a lightweight HTTP client for making asynchronous requests. This was mainly used to get familiar with including external libraries in HTML pages and interacting with web-based data sources. The hands-on experience with `axios` helped me understand how JavaScript modules can be integrated into static pages.

XState

`XState` was the most significant library I used outside of `hiphop.js`. It provides a powerful and declarative API for modeling Finite State Machines (FSMs) and statecharts in JavaScript. Several of the FSM-based examples in this project rely entirely on `XState` to define transitions, states, and actions, which allowed me to focus on state logic rather than building control structures from scratch. This was instrumental in the suite of FSM pages included in the project.

Luxon

For time-related features such as the world clock page, I used `Luxon`, a modern JavaScript date/time library that builds on the native `Intl` API. Luxon simplified the process of formatting dates, handling time zones, and performing time-based calculations.

Chapter 4

Learning HTML, CSS, and JavaScript

4.1 Motivation and Background

Although this project ultimately focused on reactive programming with `hiphop.js`, a significant portion of the work required learning basic web development skills—including HTML structure, CSS styling, and JavaScript integration—due to my initial lack of experience in these areas.

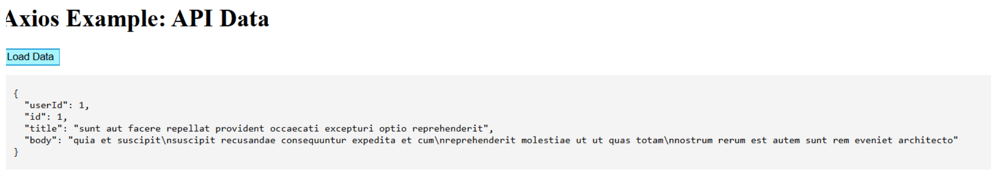
My only prior exposure to web programming had been through a project using Django (Python), which was oriented around remote router access. In that context, most of the web-related boilerplate was already provided or abstracted away. As a result, I entered this project with a need to learn web programming almost from scratch, particularly in terms of how to structure and serve static webpages, import libraries, and manage interactivity with client-side JavaScript.

4.2 Starting with Simple Examples

I initially attempted to port ABRO examples into webpages directly but ran into serious compatibility and integration issues. These early failures motivated me to step back and work through simpler HTML/CSS/JS examples designed to isolate key web concepts—specifically, module imports and dynamic JavaScript behavior.

Axios: API Fetching and Import Maps

One of the first minimal examples I created used the `axios` library to fetch data from a public API. This example helped me understand how to use `importmap` to bring in ES modules from a CDN and how asynchronous operations and DOM manipulation work within a browser context.



```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"
}
```

Figure 4.1: Axios Example — API data fetched and displayed dynamically

The following snippet shows how Axios was imported using a CDN link within an `importmap` block in the `<head>`:

```
<script type="importmap">
{
  "imports": {
    "axios": "https://cdn.jsdelivr.net/npm/axios@1.4.0/dist/esm/axios.js"
  }
}
</script>
```

This format, combined with using a `<script type="module">`, allowed me to load and use modern libraries in a standards-compliant way inside a basic HTML page.

XState: A Minimal Finite State Machine

The second example I created used the `xstate` library to implement a basic two-state toggle machine in the browser. This project was particularly important as it laid the foundation for the more complex FSM-based pages that would follow later in the project.

Just like with Axios, I used a CDN-based import of the `xstate` package through an `importmap`. The state transitions were handled by a simple interpreter connected to the DOM, allowing the displayed state to change interactively as the user clicked a button.

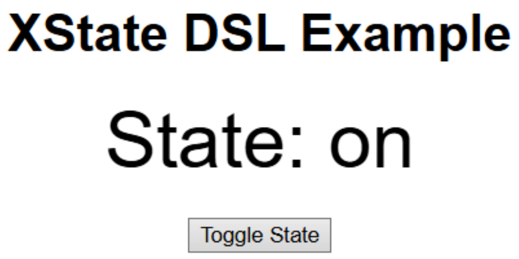


Figure 4.2: XState Example — Toggle between "on" and "off" states

These initial exercises gave me crucial hands-on experience with:

- Structuring HTML documents from scratch
- Including and using third-party libraries
- Working with JavaScript modules in the browser
- Adding basic styling with embedded CSS

They also provided a clearer sense of how to debug browser-based code and how the browser event loop and reactivity mechanisms differ from synchronous programming models.

4.3 Drawing and Interacting with FSMs in the Browser

After gaining basic familiarity with HTML and JavaScript modules through the Axios and XState examples, I moved on to deeper experimentation that would help me understand two critical areas of web development: SVG graphics and DOM interactivity.

To do this, I built a finite state machine (FSM) simulator entirely in the browser using XState for logic, SVG for visualization, and native HTML/JavaScript for interaction. This allowed me to integrate multiple learning goals: building visual diagrams with CSS and SVG, managing state transitions, and controlling behavior through time-based and event-based signals.

Automatically-Driven FSM

The first version of the simulator featured a three-state system where state transitions occurred automatically every two seconds according to a predefined signal sequence. Each state was represented by a circle, and transitions were visually indicated using SVG lines and labels. The currently active state was highlighted in green via CSS, and the text display above the SVG diagram was synchronized with the FSM's internal state.

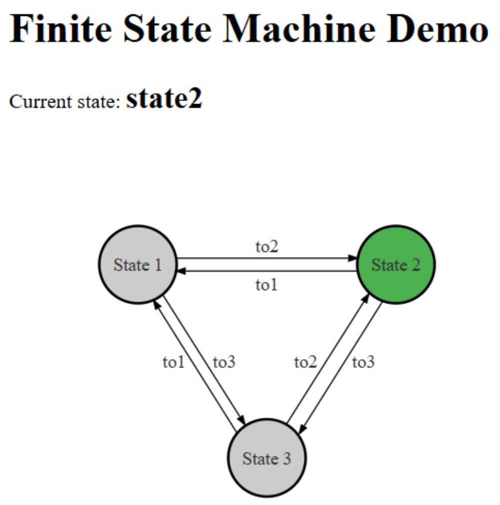


Figure 4.3: Auto-advancing FSM with three states, implemented with XState and SVG

Manual FSM Control with Buttons

Building on the automatic FSM version, I created a second version where state transitions were manually triggered via buttons. Each button sent a specific signal (`to1`, `to2`, `to3`) to the FSM when clicked. This extended my understanding of event handling and how to set up click listeners in JavaScript to directly influence program behavior.

This version retained the same FSM logic and SVG diagram, but introduced a clean separation between the visual rendering and user interaction logic. This separation helped clarify how to write modular, maintainable JavaScript code and encouraged me to think more about the architecture of reactive interfaces.

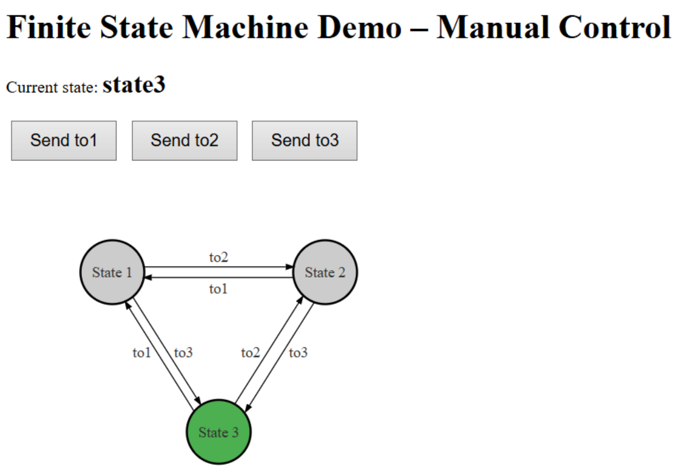


Figure 4.4: Button-controlled FSM: each button triggers a specific state transition

4.4 Porting ABRO Reactive Programs to the Browser

After working through foundational web technologies and FSM demos, I was finally able to successfully port some of the core HipHop.js ABRO examples to the browser. This required connecting button-based input events from HTML interfaces to the HipHop reactive machine running in JavaScript.

Two versions of the ABRO pattern were implemented: one based on release semantics, and the other based on reset semantics, each exposing subtle differences in the way signals are handled between discrete logical instants.

ABRO with `do/every` Release Mechanism

The first version is a browser implementation of the `release_abro` module described earlier in the report(see Appendix A.4). The counter increments only if both `A` and `B` are received after a `R` signal has occurred. This demonstrates the semantics of preemption using a `do {...} every(R)` construct, where a new release resets the waiting condition.

This webpage is a faithful visualization of the synchronous reactive logic behind the module, and provides a clear contrast with the looping version

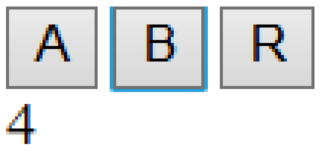


Figure 4.5: ABRO release example — counter only increments after R, A, and B are received

discussed next.

Looping ABRO with Reset Behavior

The second version, `reset_abro`, mimics a looping behavior where the counter increases automatically each time both `A` and `B` are received. A separate `Reset` button is used to reset the counter to zero. This variant highlights how looping control structures can continuously synchronize over multiple logical instants(see Appendix A.5).

Two implementations were created for this reset version:

- **Clockless version:** sends a HipHop reaction immediately upon any button press, passing in the current state of the input signals. This version is straightforward and reflects a minimal interaction loop.
- **Clock-based version:** called `counter_reset_clock`, this version introduces a fixed timer that triggers the HipHop machine at regular intervals. It collects the pressed signals since the last clock tick and sends them together in a single reaction. This better showcases the principles of synchrony and batching of events, and forms the basis for future time-based programs like the alarm.

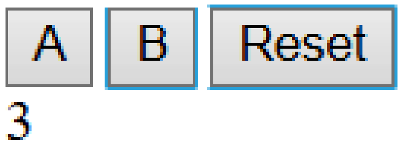


Figure 4.6: Looping ABRO counter with reset — demonstrating signal grouping across clock pulses

These ported ABRO examples served as the technical and conceptual bridge between the early button-driven experiments and the more complex synchronous applications (e.g., clocks, alarms) that followed later in the project.

Common Pitfalls When Porting HipHop.js to Webpages

While porting HipHop.js modules to run within a browser environment, I encountered two significant issues that took substantial time to resolve, both stemming from mismatches between Node.js expectations and browser limitations.

1. Import Map Configuration Initially, I used an incorrect import map, which led to unresolved module errors when trying to import `@hop/hiphop`. The correct way to configure the import map is to explicitly reference the local `hiphop-client.mjs` file:

```
<script type="importmap">
  {
    "imports": {
      "@hop/hiphop": "/hiphop-client.mjs"
    }
  }
</script>
```

Failing to provide this path results in browser errors because it tries to resolve the package as if it were installed from npm or available globally—neither of which applies here.

2. "`process is not defined`" Error Another frustrating issue was the runtime error:

```
Uncaught ReferenceError: process is not defined
    at net.js:31:25
```

This occurs because some internal parts of HipHop.js (or its dependencies) reference `process.env`, a global object available in Node.js but undefined in a browser. To bypass this error during experimentation, I added a simple browser-compatible definition of `process` in the HTML header:

```
<script>
  window.process = { env: {} };
</script>
```

While this fix is more of a workaround than a long-term solution, it enabled the scripts to load and function for my use case.

These issues highlight how HipHop.js is primarily designed with a Node.js context in mind and that extra care is needed when adapting it to a frontend environment.

4.5 Clock-Based Alarms Using `hiphop.js`

As the final part of my project, I developed a series of alarm applications that build upon the ideas of signal-driven reactivity and synchronous control. These alarms rely on a HipHop.js module that reacts to periodic clock pulses and user inputs to determine whether the alarm should be ringing. This marked the most advanced use of the synchronous logic model within a web interface, combining time-based triggers, button controls, and visual/audio feedback.

Basic Alarm Logic

The core structure of each alarm page involves a timer that periodically sends a signal to the HipHop module. This signal represents a clock pulse, along with any additional input signals (e.g., `snooze`, `stop`). The HipHop module returns a boolean output that indicates whether the alarm should be active or not.

The most minimal version of this system is simply labeled `alarm`. When active, it displays the word "Alarm" in red text. If the alarm is not ringing, the word appears in black. Below the text are two buttons for user interaction: `Snooze` and `Stop`.

Sound-Enhanced Alarm

In the second version, named `sound_alarm`, a beep sound is added for additional feedback. This version improves user perception by blinking the word "Alarm" in red and playing the beep every time the clock ticks and

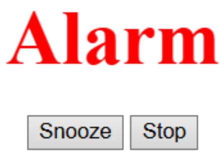


Figure 4.7: Minimal alarm interface with snooze/stop buttons and color-coded alert

the HipHop module determines that the alarm should be active. The core control logic remains the same, but the added audio and animation enhance clarity and urgency.

Prettier Alarm with Clock Display

The final iteration of this suite, called `prettier_alarm`, incorporates a more polished UI. It displays the current time in large digits instead of the word "Alarm" and uses enlarged buttons for `Snooze` and `Stop`. This version was inspired by commercial clock interfaces and merges the reactive alarm logic with a real-time clock display, built using the Luxon library.

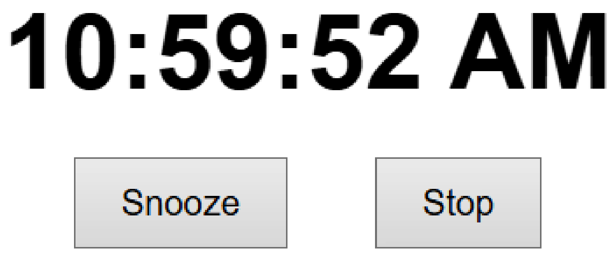


Figure 4.8: Final alarm version — large clock, refined buttons, and synchronized HipHop logic

These three alarms demonstrate how the HipHop.js model of synchrony and reactivity can be integrated into real-world interfaces that combine visual feedback, user input, and time-sensitive behavior. The transition from logic-only modules to complete user-facing pages was a key learning milestone in the project.

Chapter 5

Finite State Automata and Their Relevance

5.1 Motivation and Theoretical Foundations

A significant part of this project involved exploring the connections between reactive programming and formal models of computation, particularly finite state automata (FSA). My interest in FSAs stemmed both from prior exposure to their theoretical underpinnings and their structural similarity to synchronous reactive languages like Esterel and HipHop.js, where signal propagation can be modeled as transitions between well-defined states.

To deepen this understanding, I was directed by my supervisor, Professor Sanjiva Prasad, to study a set of materials authored by Professor Madhavan Mukund. His presentation *"Distributed Automata from Global Specifications"* provided a formal introduction to finite-state automata, distributed alphabets, and the theoretical machinery behind direct products and synchronized automata, and discussed how global behaviors can be synthesized into local automata components.

This material was crucial in helping me frame both the ABRO examples in HipHop.js and the state machine pages I developed later in the project. Concepts such as the distributed alphabet, local transitions, and coordination between components have direct parallels in the way reactive modules behave in synchrony and how input signals cause transitions in my JavaScript/XState-based implementations.

For more details on the underlying theory, see the presentation:

https://www.cmi.ac.in/~madhavan [6].

5.2 Exploring Product Automata Through Binary Counters

To deepen my practical understanding of finite state automata (FSA) and their compositional properties, I implemented a series of visual examples simulating binary counters. These were directly inspired by a previous electronics course, where we modeled binary counters using JK flip-flops, and by the idea of automata products introduced in the theoretical resource studied earlier [6].

Concept and Objective

The central idea was to simulate three bistate automata—one per flip-flop—that could act independently or in coordination. Each flip-flop can be modeled as a two-state automaton transitioning between 0 and 1. By defining different sets of input signals for each flip-flop, we can control their behavior individually or combine them to behave like a cohesive binary counter.

This gave rise to two visual demonstrations:

- One where each flip-flop behaves independently and is triggered by its own local alphabet.
- Another where they share overlapping signal sets and form a synchronized product automaton that increments a binary counter.

Independent Flip-Flop View

In the first visualization, the three flip-flops appear side by side but are not connected in any meaningful way. Each FSM responds to specific signals:

- Flip-Flop 1 accepts only `s3`
- Flip-Flop 2 accepts `s2` and `s3`
- Flip-Flop 3 accepts `s1`, `s2`, and `s3`

Each panel has separate controls to trigger its transitions. This version helps observe how localized transition functions operate and how global coordination does not emerge when input sets remain isolated.

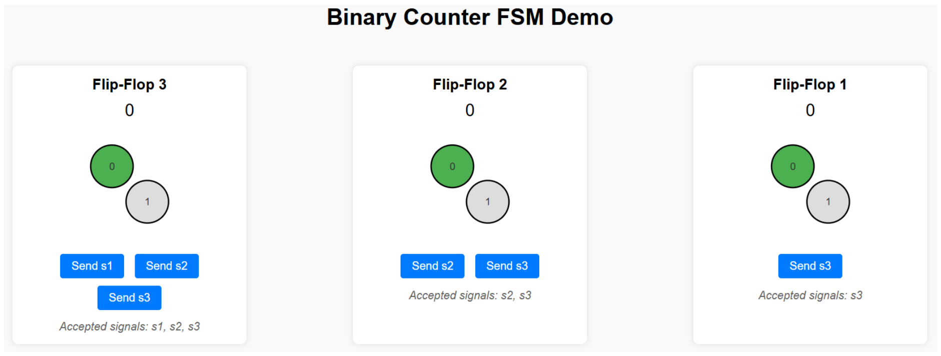


Figure 5.1: Independent flip-flop automata responding to different signal subsets

Synchronized Binary Counter (Product Automaton)

In the synchronized version, the FSMs act as components of a single composite automaton. Their transition alphabets are deliberately overlapped to form a cascading trigger pattern:

- `FSM1` accepts `s1`, `s2`, and `s3`
- `FSM2` accepts `s2`, `s3`
- `FSM3` accepts only `s3`

By sending `s3` repeatedly, the lowest-order bit toggles. Once it returns to 0, `s3` also toggles the middle bit (`FSM2`), and so on. This models the natural ripple-carry behavior of binary counters. A live counter at the bottom of the interface displays the current binary value and its decimal equivalent.

This demonstration was highly illustrative of automata product theory, showing how shared signals create coordination, and how state combinations can be used to compute or encode structured outputs. It also demonstrated a meaningful real-world application of FSMs beyond abstract control logic.

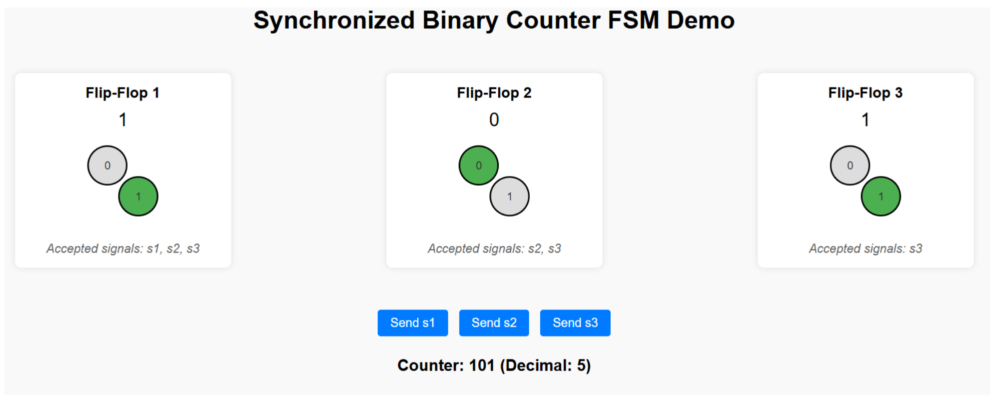


Figure 5.2: Product automaton version—flip-flops synchronize to form a binary counter

5.3 Interactive FSM Keypad Interfaces

As a final exploration of Finite State Automata, I developed a series of interactive graphical FSMs that are controlled by a virtual keypad. These examples illustrate both simple and more advanced FSM designs and demonstrate how such automata can be directly manipulated and visualized in a browser context.

5.3.1 Three-State Keypad FSM

The first example consists of a three-state machine with states A, B, and C, where:

- Pressing `0` from state A leads to state B,
- Pressing any digit `1-9` from state A transitions to C,
- From C, pressing any digit (`0-9`) leads to a self-loop on C,
- States B and C are terminal except for C's self-loop.

Each keypress is visually represented by an arrow highlight in the SVG diagram, reinforcing the association between the input and the state transition. Buttons are dynamically enabled/disabled depending on the valid transitions from the current state.

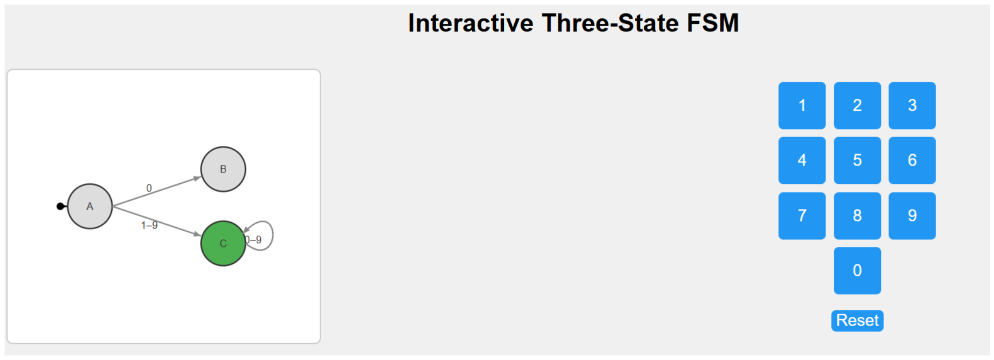


Figure 5.3: Interactive three-state FSM with input-restricted transitions.

5.3.2 FSM with Error Handling

To explore invalid input management, the FSM was extended with an error state E. Any invalid transition (e.g., pressing `0` from state B) causes the FSM to transition to E, which is visually represented by a red state. This example demonstrates the use of a formal "error sink" state, a useful concept in both software design and formal methods to catch unexpected behavior.

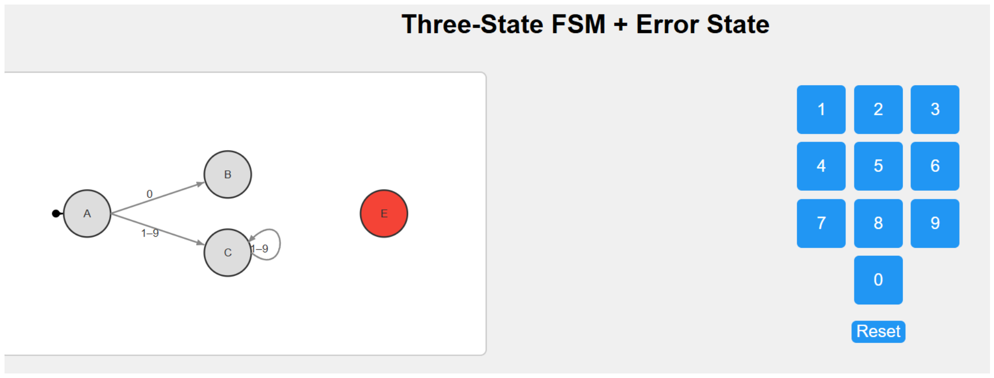


Figure 5.4: FSM with explicit error handling via a red sink state E.

5.3.3 Numeral System Aware FSM

The final example expands the same FSM structure but supports three numeric modes: binary, decimal, and hexadecimal. The input keypad and transition labels dynamically adapt to the selected numeral system. Furthermore, a counter value is constructed in the selected base as the user presses keys. This example demonstrates:

- A dynamic FSM interface driven by mode-dependent logic.
- Integration of numeric systems in user interaction.
- Real-time computation based on state and input (e.g., accumulating digits).

This highlights the pedagogical and interactive potential of FSMs in web-based environments.

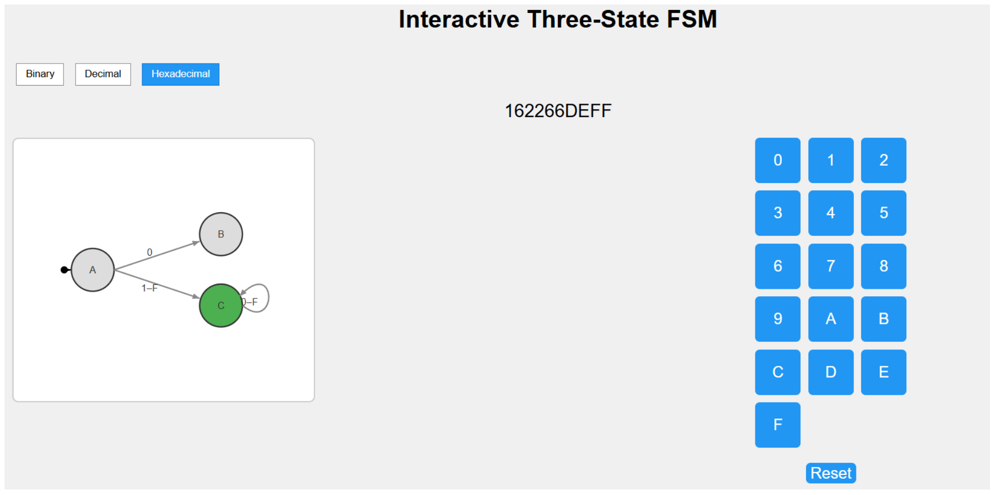


Figure 5.5: Mode-switching FSM interface supporting binary, decimal, and hexadecimal input.

Relevance and Educational Value

These interfaces were particularly useful in understanding how abstract state models can be implemented and made visible. They also provided a direct way to reinforce theoretical FSM concepts such as:

- State transitions,
- Error states and catch-all logic,

- Parameterized input alphabets,
- Stateful computation based on FSM structure.

They served as both didactic tools and creative exercises to solidify my understanding of FSM theory and browser-based JavaScript integration.

Chapter 6

Clock Utility Suite: Alarms, Timers, and World Clock

This final chapter presents a suite of time-related web applications inspired by the typical features of a smartphone clock app. The suite includes an alarm list, a world clock, a stopwatch, and an adjustable countdown timer. While the first of these — the alarm list — was built using `hiphop.js`, the others were implemented using standard JavaScript.

The reason for this separation is rooted in practical experience: the alarm list was a natural continuation of the HipHop alarm module experiments described earlier in this report. It was well-suited to reactive signal-based design, with periodic CLOCK ticks, reset logic, and clear event-driven behavior. However, when it came to implementing the stopwatch, world clock, and adjustable timer, using `hiphop.js` felt less intuitive. Rather than improving clarity or functionality, it often added unnecessary complexity. In these cases, traditional JavaScript approaches proved to be more efficient and maintainable. As such, this chapter illustrates a balanced use of tools: `hiphop.js` where it fits, and conventional JavaScript where it excels.

6.1 Alarm List: `frequency_list_alarm`

The first tool in this suite is the `frequency_list_alarm`, a multi-alarm interface that emulates the behavior of alarm apps on smartphones. This is in direct continuation of my previous work on alarms using hiphop orchestration. This program allows users to configure multiple alarms through an intuitive graphical interface. Each alarm can be customized with the following properties:

- **Time:** the exact time at which the alarm should ring.
- **Label:** an optional custom name to describe the alarm.
- **Snooze option:** a checkbox to enable or disable snooze functionality.
- **Sound choice:** the alarm sound can be selected from three available options: Beep, Bell, and Ding.
- **Frequency:** the alarm can either ring *once* or be configured to ring *daily.*

When an alarm's time is reached, it activates the corresponding HipHop reactive module. If snooze is enabled, a snooze button appears and postpones the alarm briefly upon interaction. A stop button is always available to halt the alarm. Internally, each alarm is backed by a HipHop reactive machine defined by the `AlarmSpec` module, which manages the control flow using signals like `CLOCK`, `SNOOZE`, `STOP`, and `RESET`.

This modular setup not only mimics the real-world alarm experience but also demonstrates a scalable architecture where each alarm operates independently yet reacts to a global time clock. The reactive model simplifies state transitions, especially around conditions like repeated daily activation or snooze delays.

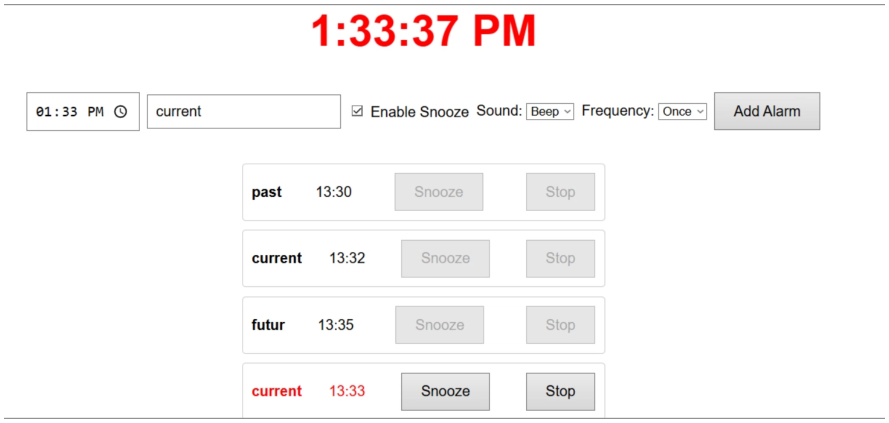


Figure 6.1: Screenshot of the `frequency_list_alarm` interface showing multiple configured alarms with snooze and stop options.

This tool also served as a learning milestone in coordinating multiple concurrent reactive processes and maintaining their lifecycle in the browser. The clock signal is dispatched every second to all alarm modules, while each module decides whether to react, snooze, or remain idle based on its state and configuration.

6.2 Stopwatch and World Clock

As part of extending the functionality of time-based utilities beyond alarms, I developed two additional standalone tools: a stopwatch and a world clock.

6.2.1 Stopwatch

The stopwatch application provides a clean interface for measuring time intervals, featuring start/stop functionality, lap recording, and a reset mechanism. Each lap is calculated based on the delta from the previous lap, and the results are dynamically listed below the timer.

The core timing logic is based on JavaScript's `Date.now()` and `setInterval`, ensuring millisecond accuracy while keeping the interface responsive. Styling is delegated to an external CSS file to maintain code clarity and separation of concerns—this proved beneficial especially when adjusting layout and appearance without touching the HTML or JavaScript.

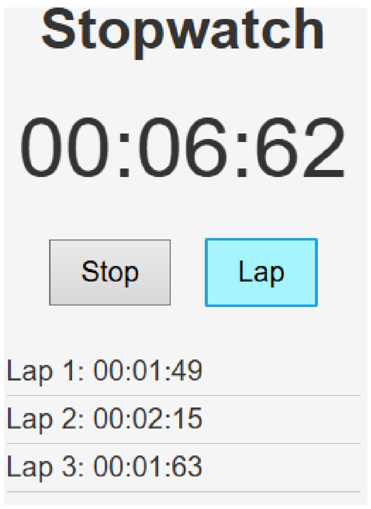


Figure 6.2: The stopwatch interface with multiple lap records.

6.2.2 World Clock

The world clock utility allows users to track the current time across multiple time zones. This feature is especially useful in globally connected settings, such as remote teams or coordination with international partners.

To manage time zone complexity, I used the `luxon` JavaScript library, a modern alternative to `moment.js`. `Luxon` simplifies conversion, formatting, and time zone adjustments by offloading the logic to a robust and tested library. Users can select any IANA time zone from a searchable dropdown powered by the browser's `Intl.supportedValuesOf` API, with support for keyboard navigation and filtering.

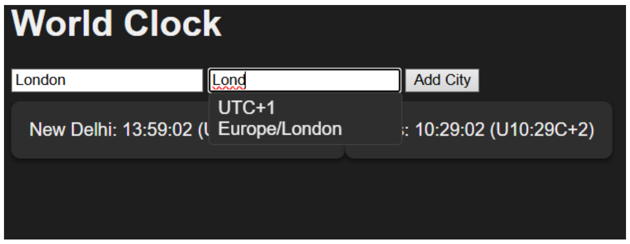


Figure 6.3: World clock application displaying multiple cities and their current times.

The use of a centralized stylesheet and modular JavaScript (imported via ES modules) reflects good development practices adopted as the project went on. This not only improved maintainability but also made these components adaptable for future reuse or integration.

6.2.3 Adjustable Multi-Alarm Timer

The final tool in this suite is the `Multi-Alarm Timer`, which was designed to offer significantly more flexibility than a typical timer application. The primary goal was to support use cases where a single end-time alarm is insufficient, such as workouts or timed lectures that benefit from intermediary alerts.

Users can define:

- A total duration per loop.
- The number of repetitions (loops) of this timer.
- Any number of intermediary alarms within each loop.

The timer interface is divided into four key sections: duration and loop configuration, live time display with loop and next-alarm indicators, controls (start and reset), and a section for adding intermediary alarms. A short preparatory countdown is also included before the first loop begins.

One pedagogical example is for a professor who sets a 60-minute timer to mark the end of a lesson, but adds a 15-minute warning alarm for initiating an in-class quiz. Another practical use case is interval training: a user wanting to alternate 30 seconds of activity with 5 seconds of rest across 12 rounds could configure a 35-second timer with a 30-second intermediary alarm and loop it 12 times.

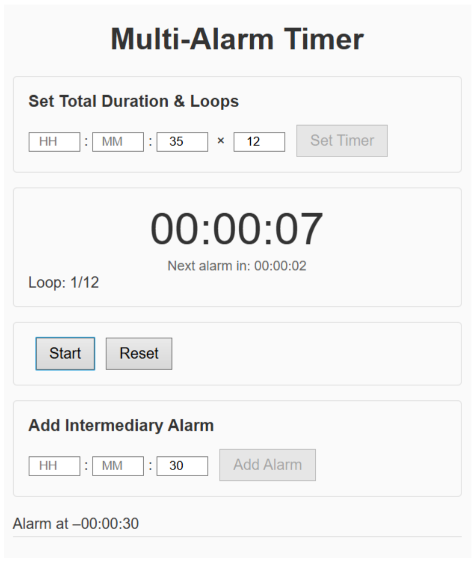


Figure 6.4: The adjustable timer running with 12 loops of 35 seconds and one intermediary alarm at 30 seconds.

From a development perspective, this timer required precise time tracking, multiple audio cues, and UI updates for each state transition. Alarms are stored and sorted dynamically, and all sounds are preloaded to minimize delay. The separation between configuration and execution phases ensures the timer is both robust and user-friendly.

Chapter 7

Challenges and Lessons Learned

7.1 Challenges Faced

7.1.1 General Contextual Difficulties

This project was my first in-depth encounter with web development. At the outset, my knowledge of the core technologies—`HTML`, `CSS`, and `JavaScript`—was nearly nonexistent. Every implementation task became a learning opportunity, ranging from understanding basic syntax to figuring out how to link external `.js` or `.css` files, importing libraries correctly using `script` tags or import maps, and even triggering sounds in the browser environment.

The choice of `hiphop.js` as the central orchestration library introduced another layer of difficulty. While it provided a powerful model inspired by synchronous languages like Esterel, it remains a very niche tool. Documentation is limited, and much of the material I could find was outdated or incompatible with the current version. For example, constructs like `pause;` that appear in older tutorials are no longer valid, having been replaced by `yield;` and other newer syntax.

Beyond the technical challenges, the semester context made things more difficult. As a foreign student in India, adapting to a much heavier academic workload compared to what I was used to was tough. Many assignments felt like standalone mini-projects. The extreme heat, very different from my home climate, and several episodes of food poisoning added to the difficulty of maintaining consistent progress on this project.

7.1.2 Technical Obstacles and Debugging Hurdles

In addition to these broader challenges, I encountered several specific technical issue, among which:

- **Causality Errors in `hiphop.js`**: One of the most complex debugging scenarios involved causality loops—where signal definitions depended on each other in a circular fashion. These errors are often subtle and hard to trace, especially in larger reactive machines.
- **Module Import and Environment Problems**: At first, I struggled with proper import map usage and browser-specific quirks. A notable example was the "`process is not defined`" error—caused by libraries expecting a Node.js environment. The workaround of defining `window.process = { env: {} }` manually was something I only discovered after significant experimentation.
- **Audio Playback Restrictions**: Browsers restrict autoplay of audio for user experience reasons. This forced me to add a user interaction trigger (like a click event) to unlock audio, which is essential for apps like alarms and timers.

7.2 Skills and Concepts Acquired

Despite these challenges, the project was immensely rewarding and allowed me to develop several key skills and gain practical experience in multiple areas:

- **Modular Design and Code Separation**: I learned the importance of separating logic (JavaScript), presentation (CSS), and structure (HTML). This improved not only the readability of my code but also its maintainability.
- **Reactivity and Synchrony with `hiphop.js`**: The declarative and synchronous model of `hiphop.js` trained me to think in terms of signal propagation, instant reactions, and temporally structured behavior—very different from the asynchronous patterns common in JavaScript.

- **Finite State Machines in Practice**: The FSM-based interfaces I built helped reinforce formal concepts with hands-on implementations. I gained both theoretical understanding and design intuition, especially around state transitions, signal handling, and product automata.

- **Web Development Fundamentals**: While basic, I developed a surprisingly wide skill set across `HTML`, `CSS`, and `JavaScript`—covering styling, user input handling, DOM manipulation, browser API usage, and responsive UI design.

- **Effective Paper Reading Techniques**: I learned a more strategic approach to academic paper reading. Instead of reading from start to end linearly, I was advised to first read the abstract and conclusion, skim through the body to locate key concepts, and then return for a deep read—improving both comprehension and retention.

- **Project Sharing and Containerization**: I learned to set up a GitHub repository from scratch and prepare my project for deployment via Docker, which makes it easier to reproduce or share across systems.

Chapter 8

Conclusion

This project began as an exploration of reactive programming in the browser, but quickly became a much deeper learning experience—both technically and personally. With little prior knowledge in web development, I had to pick up and integrate a wide range of technologies and concepts: HTML, CSS, JavaScript, library imports, audio APIs, and browser-specific behaviors.

Working with `hiphop.js` was both rewarding and challenging. As a niche library inspired by Esterel, it pushed me to understand synchronous logic, causality constraints, and reactive orchestration in a way that regular asynchronous JavaScript does not. Implementing real-world applications like alarm systems, timers, and state-based interactions helped solidify my understanding of Finite State Machines and modular system design.

Looking back, this project helped me build confidence in tackling unfamiliar technologies, navigating tools where the documentation is hard to navigate and find, and breaking complex problems into manageable components.

There are a few things that might be taken away from this experience. First, that learning is often most effective when it is grounded in implementation. Second, that unfamiliar or niche tools can still provide valuable insights—especially when they challenge one's usual ways of thinking. And finally, that combining theory with hands-on development, even in small steps, builds both understanding and confidence.

This project did not aim to solve any large-scale problem, but it did offer an opportunity to practice integrating systems thinking, reactive modeling, and frontend development in a cohesive way. In that sense, it fulfilled its purpose: to serve as a platform for exploration and learning.

Acknowledgments

I would like to express my sincere gratitude to Professor Sanjiva Prasad for his invaluable guidance and mentorship throughout this project. His insights into synchronous programming, Esterel, and finite-state automata, as well as his support in navigating academic literature, were instrumental in shaping the direction and depth of my work.

I also gratefully acknowledge the comprehensive resources provided by Manuel Serrano[8] and the official `hiphop.js` documentation[7] maintained by the `hop.inria.fr` team, which were essential to understanding and implementing the reactive programming concepts used in this project.

Bibliography

[1] Gérard Berry and Manuel Serrano. HipHop.js: (A)Synchronous reactive web programming. In *PLDI '20 - 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 533–545, London UK, United Kingdom, July 2020. ACM.

[2] Maxime Durand. Hiphop.js and web programming project repository. `https://github.com/Maxime-cod/hiphop-and-webdev-project.git`, 2025. Accessed May 2025.

[3] Stephen A. Edwards. The esterel synchronous programming language and its application to real-time systems. `https://www.cs.columbia.edu/~sedwards/presentations/2005memocode-esterel.pdf`, 2005. Accessed May 2025.

[4] Hop.js Documentation. Hiphop control flow keywords. `http://hop.inria.fr/home/hiphop/doc/lang/flow.html`. Accessed May 2025.

[5] Jayanth Krishnamurthy and Manuel Serrano. Causality error tracing in hiphop.js. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming*, PPDP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Madhavan Mukund. Distributed automata from global specifications. Presentation at VerTeCS2 Symposium, IIT Delhi, March 2024. Accessed May 2025.

[7] INRIA Hiphop.js Manuel Serrano. Getting started with hiphop.js. `http://hop.inria.fr/home/hiphop/index.html`, 2025. Accessed May 2025.

[8] Manuel Serrano. hiphop.js: Synchronous reactive programming in javascript. `https://github.com/manuel-serrano/hiphop`, 2015. Accessed: 2025-05-09.

Appendix A

Appendix A: ABRO Program Suite

Overview

The ABRO suite is a set of small programs written using `hiphop.js` to illustrate its key reactive constructs. These programs are not complete web applications; rather, each consists of a `hiphop` module and a JavaScript file that triggers a sequence of reactions. The goal of the suite is twofold: to help me understand core ideas such as signal presence, concurrency, and logical instants, and to serve as minimal, clear reference examples for others exploring `hiphop.js`.

The name "ABRO" comes from the classical control pattern where a system waits for both `A` and `B` signals before emitting output `O`, and resets on `R`. While some variants omit `R`, all explore deterministic coordination between signals.

—

A.1 Example : Basic ABRO — Fork and Await

Concept

This first program introduces the core concept of `fork` and `await`. It waits for the signal `A` in one branch and `B` in the other, using `fork/par` to allow both waits to occur concurrently. Once both signals have been received, the output O is emitted. This demonstrates basic parallel composition and synchronization.

HipHop Module Code

```
import { ReactiveMachine } from "@hop/hiphop";

export const abo = hiphop module() {
   in A;
   in B;
   out O = 0;

   fork {
     await (A);
   } par {
     await (B);
   }
   emit O(O.preval + 1);
}
```

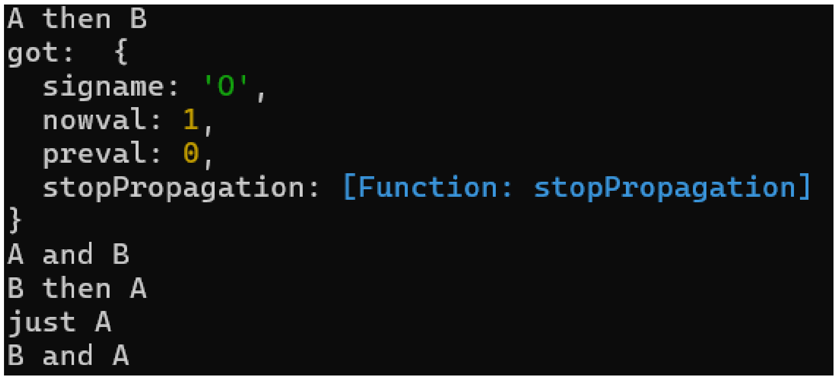
Main JavaScript Driver

```
import { ReactiveMachine } from '@hop/hiphop';
import { abo } from './abo.mjs';

const m = new ReactiveMachine(abo);

m.addEventListener("O", e => console.log("got: ", e));

m.react({});
console.log("A then B");
m.react({ A: 1 });
m.react({ B: 2 });

console.log("A and B");
m.react({ A: 3, B: 4 });

console.log("B then A");
```

```
m.react({ B: 5 });
m.react({ A: 6 });

console.log("just A");
m.react({ A: 7 });

console.log("B and A");
m.react({ B: 8, A: 9 });
```

Observed Output

The output below shows how the module reacts only after both `A` and `B` are present. The signal `O` is emitted after the second of the two is received, and the value increments with each completion.



```
A then B
got:  {
  signame: 'O',
  nowval: 1,
  preval: 0,
  stopPropagation: [Function: stopPropagation]
}
A and B
B then A
just A
B and A
```

Figure A.1: Console output from the basic ABO example

A.2 Example : Looping ABO — Continuous Synchronization

Concept

This variation builds upon the basic ABRO pattern by wrapping it in a `loop` block. After both `A` and `B` are received and the output `O` is emitted, the module resets and waits again. This introduces the concept of continuous synchronization and automatic reset within the same program structure.

HipHop Module Code

```
import { ReactiveMachine } from "@hop/hiphop";

export const loop_abo = hiphop module() {
   in A;
   in B;
   out O = 0;

   loop {
     fork {
       await (A.now);
     } par {
       await (B.now);
     }
     emit O(O.preval + 1);
   }
}
```

Observed Out

Each time both signals `A` and `B` are received (in any order and not necessarily during the same reaction), the output `O` is emitted and the module goes back to the beginning immediately to await the next pair. This creates a cyclic, reactive synchronization that mirrors classical ABO patterns used in control systems.

A.3 Example : ABCO — Parallel Synchronization of Three Signals

Concept

This variation of the ABRO pattern introduces a third signal, `C`, to test whether `hiphop.js` can handle more than two concurrent threads. The module waits for `A`, `B`, and `C` in parallel using three `await` statements within a `fork/par` structure. Once all three signals are present, the module emits the output signal `O`.

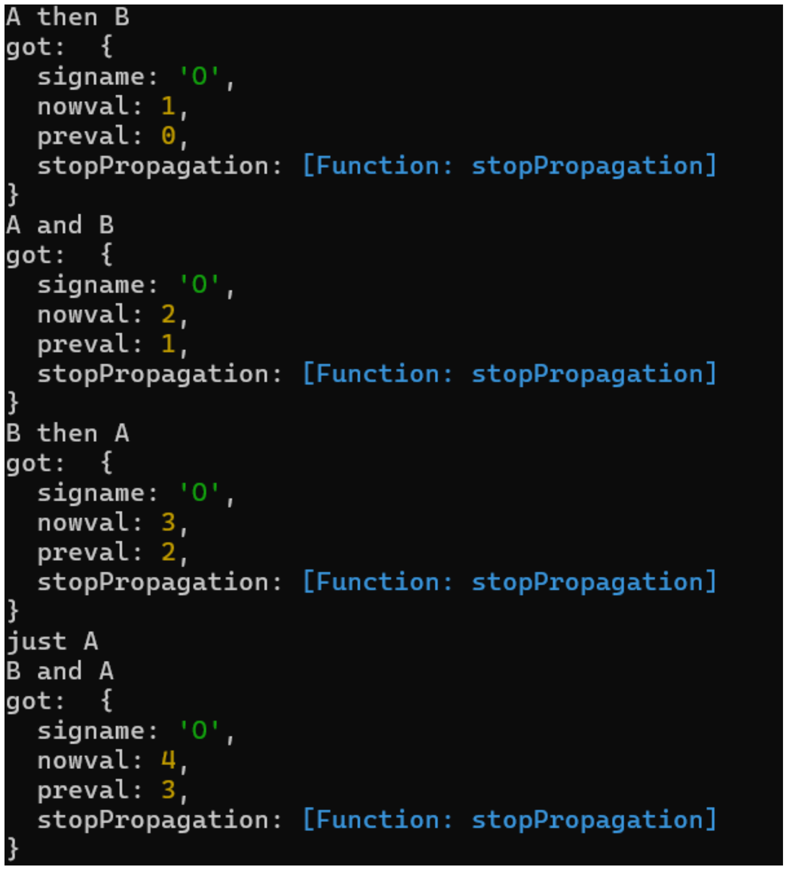

```
A then B
got:  {
  signame: 'O',
  nowval: 1,
  preval: 0,
  stopPropagation: [Function: stopPropagation]
}
A and B
got:  {
  signame: 'O',
  nowval: 2,
  preval: 1,
  stopPropagation: [Function: stopPropagation]
}
B then A
got:  {
  signame: 'O',
  nowval: 3,
  preval: 2,
  stopPropagation: [Function: stopPropagation]
}
just A
B and A
got:  {
  signame: 'O',
  nowval: 4,
  preval: 3,
  stopPropagation: [Function: stopPropagation]
}
```

Figure A.2: Output of the looping ABO module

HipHop Module Code

```
import { ReactiveMachine } from "@hop/hiphop";

export const abco = hiphop module() {
   in A;
   in B;
   in C;
   out O = 0;

   fork {
     await (A.now);
```

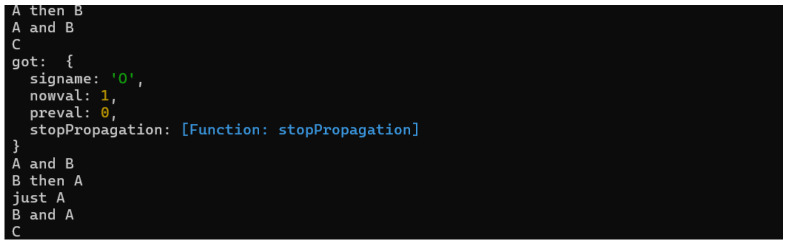
```
  } par {
    await (B.now);
  } par {
    await (C.now);
  }
  emit O(O.preval + 1);
}
```

Note on Initialization

It is important to trigger an initial empty reaction (`m.react({})`) right after creating the reactive machine. Without this, the first reaction may not be registered correctly, leading to confusing or inconsistent results. This initialization step ensures that the internal state of the machine is properly set before processing input signals.

Observed Output

The program emits `O` only after all three signals—`A`, `B`, and `C`—have been received. This demonstrates that multiple parallel branches can be used within a single `fork` structure, and that synchronization scales beyond the basic two-signal ABO pattern.



```
A then B
A and B
C
got:  {
  signame: 'O',
  nowval: 1,
  preval: 0,
  stopPropagation: [Function: stopPropagation]
}
A and B
B then A
just A
B and A
C
```

Figure A.3: Output of the ABCO module showing synchronization of three signals

A.4 Example : Release ABRO — Periodic Synchronization via `every`

Concept

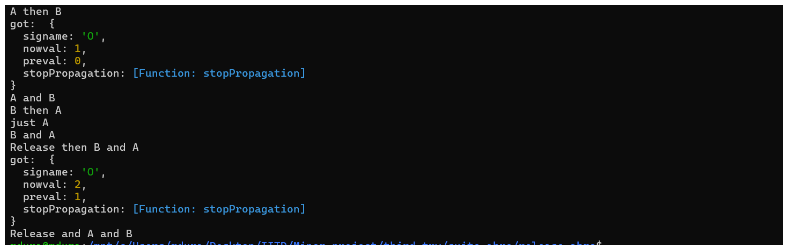
This version introduces the `every` construct to create a repeating ABRO pattern, but with external control over when it resets. The module waits for signals `A` and `B` and emits output `O` when both are received. However, unlike a `loop`, the ABRO block only restarts when the signal `R` is received. This shows how `hiphop.js` can be used to structure recurring logic triggered by external "release" conditions.

HipHop Module Code

```
import { ReactiveMachine } from "@hop/hiphop";

export const release_abro = hiphop module() {
   in A;
   in B;
   in R;
   out O = 0;

   do {
     fork {
       await (A.now);
     } par {
       await (B.now);
     }
     emit O(O.preval + 1);
   } every (R.now)
}
```

Observed Output

The program waits for both `A` and `B`. Once both are received, it emits `O`, and then does nothing until a new `R` signal arrives. This introduces controlled periodic synchronization—useful when a reactive system should reset based on an external pulse or cycle.



```
A then B
got:  {
  signame: 'O',
  nowval: 1,
  preval: 0,
  stopPropagation: [Function: stopPropagation]
}
A and B
B then A
just A
B and A
Release then B and A
got:  {
  signame: 'O',
  nowval: 2,
  preval: 1,
  stopPropagation: [Function: stopPropagation]
}
Release and A and B
```

Figure A.4: Output of the Release ABRO module showing synchronization gated by signal `R`

A.5 Example : ABRO with Strong vs. Weak Abort

Concept

This pair of programs compares the behavior of `abort` and `weakabort` in `hiphop.js`. Both modules contain an infinite loop that emits `O` each time signals `A` and `B` are received. However, the two differ in how they respond to a third signal, `R`, which interrupts execution.

- `abort` (strong preemption): terminates the block *immediately*, without completing the current reaction.
- `weakabort`: lets the current reaction finish, then aborts before the next.

By running the same input sequence against both modules in parallel, we can observe how timing of `R` influences whether `O` is emitted.

HipHop Module Code — Weak Abort

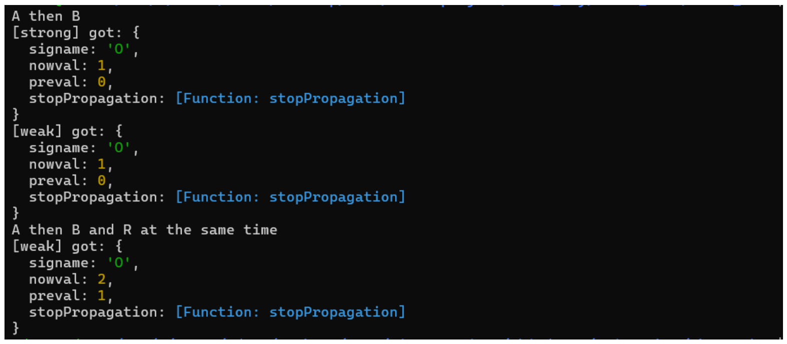
```
import { ReactiveMachine } from "@hop/hiphop";

export const weakabort_abro = hiphop module() {
   in A;
   in B;
   in R;
   out O = 0;

   weakabort(R.now){
     loop {
       fork {
         await (A.now);
       } par {
         await (B.now);
       }
       emit O(O.preval + 1);
     }
   }
}
```

HipHop Module Code — Strong Abort

```
import { ReactiveMachine } from "@hop/hiphop";

export const strongabort_abro = hiphop module() {
   in A;
   in B;
   in R;
   out O = 0;

   abort(R.now){
     loop {
       fork {
         await (A.now);
       } par {
         await (B.now);
       }
       emit O(O.preval + 1);
     }
   }
}
```

Observed Output

When executed in parallel with the same sequence of reactions, the strong abort module will cancel the loop *before* it gets to emit `O` if signal `R` is received during the same instant. In contrast, the weak abort module completes the emission before canceling the loop.



```
A then B
[strong] got: {
  signame: 'O',
  nowval: 1,
  preval: 0,
  stopPropagation: [Function: stopPropagation]
}
[weak] got: {
  signame: 'O',
  nowval: 1,
  preval: 0,
  stopPropagation: [Function: stopPropagation]
}
A then B and R at the same time
[weak] got: {
  signame: 'O',
  nowval: 2,
  preval: 1,
  stopPropagation: [Function: stopPropagation]
}
```

Figure A.5: Side-by-side output comparison of strong and weak abort behavior

A.6 Example : Preemption with `do ... every` — Interruptible Sequence

Concept

This example demonstrates how a sequence of signal-based steps can be preempted and restarted using the `do ... every` construct. The module emits an initial value, then waits for a series of `A` signals to increment a counter, emitting signal `O` after each one. However, if signal `R` is received before the sequence completes, the execution inside the `do` block is immediately interrupted and restarted. This is useful for modeling interruptible behavior like retries, resets, or external overrides.

HipHop Module Code

```
import { ReactiveMachine } from "@hop/hiphop";
```

```
export const every_preemption_aro = hiphop module() {
   in A;
   in R;
   out O = 0;

   do {
     emit O(0);
     await(A.now);
     emit O(O.preval + 1);
     await(A.now);
     emit O(O.preval + 1);
     await(A.now);
     emit O(O.preval + 1);
     await(A.now);
     emit O(O.preval + 1);
   } every (R.now)
}
```

Observed Output

The program emits a zero and waits for signal `A` four times, incrementing the counter each time. When signal `R` is received:

- If the full sequence has finished, the loop restarts normally.
- If it hasn't, the current sequence is interrupted and reset.

This is clearly seen in the output: when `R` is triggered after only two `A` signals, the sequence resets early. When `A` and `R` arrive in the same instant, the restart takes priority after the current reaction.

Console Output

```
got: { signame: 'O', nowval: 0, preval: 0, ... }
Trigger A more time than the max counter
got: { signame: 'O', nowval: 1, preval: 0, ... }
got: { signame: 'O', nowval: 2, preval: 1, ... }
got: { signame: 'O', nowval: 3, preval: 2, ... }
got: { signame: 'O', nowval: 4, preval: 3, ... }
```

```
release with R (loop again the module)
got: { signame: 'O', nowval: 0, preval: 4, ... }
trigger A only two time before releasing with R
got: { signame: 'O', nowval: 1, preval: 0, ... }
got: { signame: 'O', nowval: 2, preval: 1, ... }
got: { signame: 'O', nowval: 0, preval: 2, ... }
trigger A two times again then trigger A and R at the same time
got: { signame: 'O', nowval: 1, preval: 0, ... }
got: { signame: 'O', nowval: 2, preval: 1, ... }
got: { signame: 'O', nowval: 0, preval: 2, ... }
```

A.7 Example : Preemption with `abort` — Reaction Boundaries

Concept

This example explores the use of `abort` to model preemption at a fine-grained level. The module waits for signal `A`, then proceeds through a sequence of steps, emitting `O` with incrementing values. The 'yield' keyword is used to explicitly define reaction boundaries—each 'yield' splits execution into separate logical instants. If the signal `R` is received at any point before the sequence completes, the `abort` block is exited, skipping the remaining steps. The surrounding `loop` restarts the behavior.

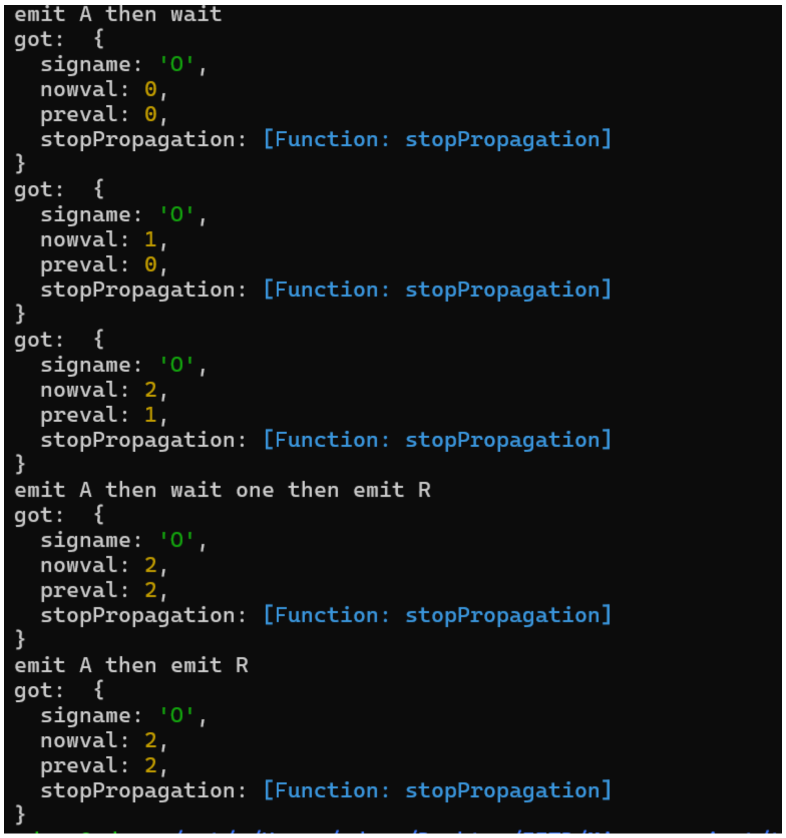
HipHop Module Code

```
import { HALT, ReactiveMachine } from "@hop/hiphop";

export const preemption_abort = hiphop module() {
  in A; in R; out O = 0;

  loop {
    abort (R.now) {
      await(A.now);
      emit O(0);
      yield;                      // ← boundary: end this reaction
      emit O(1);
      yield;                          // ← boundary: end this reaction
    }
    emit O(2);
    yield;                             // boundary again
  }
};
```

Observed Output

When A is triggered and no R occurs, the module emits O in three successive reactions: 0, 1, and finally 2. If R is triggered at any point within the abort block (before reaching O(2)), the module exits early and skips the remaining emissions from the block. The loop then begins a new cycle, ready to await A again.

This construct is particularly useful for designing reactive systems with time-sensitive interruption conditions, such as emergency stops or user cancellations.



```
emit A then wait
got:  {
  signame: 'O',
  nowval: 0,
  preval: 0,
  stopPropagation: [Function: stopPropagation]
}
got:  {
  signame: 'O',
  nowval: 1,
  preval: 0,
  stopPropagation: [Function: stopPropagation]
}
got:  {
  signame: 'O',
  nowval: 2,
  preval: 1,
  stopPropagation: [Function: stopPropagation]
}
emit A then wait one then emit R
got:  {
  signame: 'O',
  nowval: 2,
  preval: 2,
  stopPropagation: [Function: stopPropagation]
}
emit A then emit R
got:  {
  signame: 'O',
  nowval: 2,
  preval: 2,
  stopPropagation: [Function: stopPropagation]
}
```

Figure A.6: output of the abro module using abort_based preemption