HipHop.js: (A)Synchronous Reactive Web Programming

Gérard Berry
Collège de France
Paris, France
Gerard.Berry@college-de-france.fr

Manuel Serrano
Inria/UCA
Sophia Antipolis, France
Manuel.Serrano@inria.fr

Abstract

We present HipHop.js, a synchronous reactive language that adds synchronous concurrency and preemption to JavaScript. Inspired from Esterel, HipHop.js simplifies the programming of non-trivial temporal behaviors as found in complex web interfaces or IoT controllers and the cooperation between synchronous and asynchronous activities. HipHop.js is compiled into plain sequential JavaScript and executes on unmodified runtime environments. We use three examples to present and discuss HipHop.js: a simple web login form to introduce the language and show how it differs from JavaScript, and two real life examples, a medical prescription pillbox and an interactive music system that show why concurrency and preemption help programming such temporal applications.

CCS Concepts: • **Software and its engineering** → **Orchestration languages**; **General programming languages**; **Language types**; **Parallel programming languages**; **Multiparadigm languages**; **Distributed programming languages**.

Keywords: Reactive Programming, Synchronous Programming, Web Programming, JavaScript

ACM Reference Format:
Gérard Berry and Manuel Serrano. 2020. HipHop.js: (A)Synchronous Reactive Web Programming. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3385412.3385984

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385984

1 Introduction

We present HipHop.js, a synchronous orchestration language for JavaScript. To help web application development, HipHop.js was built within Hop.js [35], which is itself a multitier JavaScript extension dedicated to facilitate the programming of web client-server asynchronous interactions in various ways. But the HipHop.js compiler generates plain JavaScript code that can be used in any server or client JavaScript environment, without Hop.js and without any need for an extra runtime support.

Our broad goal is to facilitate the design and programming of complex applications by smoothly integrating three computation models and programming styles that have been historically developed in different communities and for different purposes. Let us first recall these models, which were characterized in the 1980's. *i) Transformational programs* simply compute output values from input values, with comparatively simple interaction with their environment. This is the domain of classical sequential programming languages. *ii) Asynchronous concurrent programs* perform interactions between their components or with their environment with uncontrollable timing, using typically network-based communication. This domain was started by Hoare's CSP in the 1970's, and languages such as Erlang [3] or CML [33] were designed for it. Web programming in JavaScript or other languages also belongs to this category, but with an asymmetric client/server view. *iii) Synchronous reactive programs* react to external events in a conceptually instantaneous and deterministic way: examples are real-time process control, command systems, communication protocol nodes, and, more generally, temporal orchestration of complex activities. This is the domain of *synchronous languages*, started in the 1980's by Esterel [5], Lustre [21], and Signal [20], followed by many other languages and transformed into an industrial success by SCADE 6 [10] for safety-critical applications.

Asynchronous and synchronous programs all use concurrency and communication, but differently: asynchronous concurrency is inherently non-deterministic with communication taking arbitrary non-zero time; on the opposite, synchronous concurrency is deterministic with communication in conceptually zero time. Both concurrency models have found many applications, but they have remained quite dissociated. We show that integrating HipHop.js's synchrony with JavaScript's asynchrony can simplify web applications

where reactive aspects are as important as interactive ones, which we believe will become more and more frequent. To the best of our knowledge, such a smooth blending of both concurrency models in the context of a mainstream web-oriented language has never been achieved before.

JavaScript is obviously adequate for programming transformational and interactive or simple reactive systems. Its object orientation helps handling data structures and implementing classical algorithms. Its functional aspect makes it suitable for implementing basic interactions and GUIs using callbacks. But, on the asynchronous side, explicitly coding the exchange data between servers and clients remains mandatory. Hop.js removes the need for such manual communication code. Servers and clients are written in the same code, where locations of code fragments are specified by *multitier primitives* [35] in the form of annotations telling which server or client an expression belongs to. Then Hop.js automatically generates secure communication code for the necessary data transfers. In this paper, we will stick to basic usage of Hop.js, because this is not our main subject and because HipHop.js does not depend on Hop.js. All Hop.js code in our examples could be written in plain JavaScript, albeit with a significant size and complexity increase.

On the reactive side, JavaScript is based on an atomic execution mode that implements a kind of synchronous reaction. ECMAScript 2015 [15] has augmented it by some basic features that help asynchronous programming: `yield` stops the control and restarts from there on demand; promises and `async` simplify the blending of asynchronous executions into sequential programming. These additions do help writing basic temporal behaviors, but in a limited way. HipHop.js keeps them but adds far more powerful features. First, *synchronous concurrency* helps simplifying and modularizing designs, and *synchronous signaling* makes it possible to instantly communicate between synchronous concurrent statements to exchange data and coordination signals. Second, powerful event-driven reactive *preemption statements* borrowed from Esterel finely control the lifetime of the arbitrarily complex program statements to which they apply, instantly killing them when their control events occur; examples are `abort(`*event*`){...}`, `every(`*event*`){...}`, and a general label-based preemptive escape mechanism compatible with concurrency. Third, an extension of `async` that synchronously signals termination completion to concurrent statements and automatically performs automatic resource cleaning when preempted for any reason. We show that their combination makes reactive programming more powerful and flexible than with plain JavaScript, because it makes the temporal geometry of complex executions explicit instead of hidden in implicit relations between state variables. Our examples aim at illustrating the HipHop.js the programming style and show its advantages.

An important point is that we chose to design HipHop.js as a specific language, instead of a JavaScript extension as for Hop.js. This is because direct integration examples of concurrency and preemption in classical sequential languages always paid a semantic price we want to avoid. But we kept the syntax very similar, with a simple way of using plain JavaScript for any data computation.

To be executed, we compile a HipHop.js program into sequential plain JavaScript code. Concurrency, communication, and preemption are compiled away by adequately microscheduling the HipHop.js reactive statements in a way that respects the HipHop.js semantics, identical to Esterel's one. The HipHop.js generated code is embedded into a JavaScript *reactive machine* to be called from JavaScript using a simple interface. At runtime, since JavaScript's execution is atomic, the execution of each HipHop.js reaction is uninterruptible, which is key to ensure its correctness. This is a big bonus compared to compiling Esterel into C for instance, where run-time reaction atomicity has to guaranteed by the user separately for each usage, in a way that highly depends on the execution environment.

In the sequel, we focus on the HipHop.js's language design, the programming style it offers, and its tight integration with JavaScript. To illustrate the diversity of its potential applications, we present three examples: a simple login GUI to introduce the main constructs and the integration within JavaScript, a medical application related to drug prescription, and a larger scale interactive music platform used in actual concerts that makes extensive use of all the expressiveness HipHop.js has to offer.

The paper is organized as follows. In Section 2, we present two implementations of the login web page: first in JavaScript then in HipHop.js, introducing the language on-the-fly and explaining of how it modularizes and simplifies temporal programming; we complete the login implementation by the Hop.js coding of a HTML GUI, showing how we make the communication between HipHop.js, Hop.js, and HTML as simple as possible. In Section 3, we show how easy it is to extend the specification to make our application richer while reusing the unmodified initial HipHop.js code, which would be difficult in JavaScript. Section 4 is devoted to the drug prescription and interactive music examples. Section 5 briefly describes the JavaScript runtime implementation of HipHop.js programs. Section 6 discusses related work, and we conclude in Section 7.

2 HipHop.js vs. JavaScript, by the Example

We introduce HipHop.js with a simple login panel as used by many web applications. When both the user name and password are filled with at least two characters, a login button is activated. When clicking it, the user credentials are checked by sending them to a third-party server, *e.g.*, an OAuth server. The session starts when the server acknowledges login validity, which also makes a logout button appear; the user can then logout by pressing it. A local clock starts when login

is acknowledged and forces logout when timeout occurs. During an active session, clicking login causes immediate logout and restart of the login phase.

As the specification of the application is simple, JavaScript would probably be considered well adapted to implement it. Nevertheless, it is still of sufficient complexity to introduce and illustrate the main features of HipHop.js and its advantages vs. plain JavaScript. We will see more on this point in Section 3 when adding a new feature to the specification.

2.1 A JavaScript Implementation

Let us first show how the application can be programmed in JavaScript, to later emphasize the differences with the HipHop.js version presented in Section 2.2 and give a first comparison of both programming models. First, we focus on event handling, delaying the HTML part to Section 2.4 and assuming that trivial listeners with obvious names are associated with HTML elements. We use global variables (registers) to model the application's states. By convention, we start their names with 'R'.

A first JavaScript component handles the name and password inputs using three registers and three functions:

```
1 function enableLoginButton() {
2    return (Rname.length >= 2 && Rpasswd.length >= 2);
3 }
4 function nameKeypress(value) {
5    Rname = value;
6    RenableLogin = enableLoginButton();
7 }
8 function passwdKeypress(value) {
9    Rpasswd = value;
10   RenableLogin = enableLoginButton();
11 }
```

Enabling and disabling login is done by setting `RenableLogin`.

The second JavaScript component is the `authenticate` function, called-back by the GUI when the user clicks the login button when it is enabled. Using `RconnStatus` and `Rconn`, it disconnects from a possibly running session, updates the GUI to notify the ongoing authentication, and sends an authentication request to the server:

```
12 async function authenticate() {
13   let conn = ++Rconn;
14   logout();
15   RconnState = "connecting";
16   let v = await authenticateSvc(Rname, Rpasswd).post();
17   if(v && conn === Rconn) startSession();
18 }
```

The register `Rconn` and a local variable `conn` are set (line 13) to number authentication requests and detect situations where another authentication has been asked for before the current one is completed. A session that might still be active is terminated before requesting the new authentication (line 14). The connection status is updated (line 15); the request is sent over the network to the authentication server (line 16); when received, the return value is checked (line 17) to start a new session once login has been approved. Notice the use of the JavaScript reactive statements `async` and `await`.

The third component handles sessions. Using registers `RconnStatus`, `Rtime` and `Rintv`, it declares the `startSession` function invoked after a successful authentication (line 17):

```
19 function startSession() {
20    RconnState = "connected"; Rtime = 0;
21    Rintv = setInterval(function() {
22       if(++Rtime > MAX_SESSION_TIME) logout();
23       update();
24    }, 1000);
25    update();
26 }
```

The status is updated and the connection time is reset; a JavaScript timer is started to increment the clock each second (line 21) and to end the session after the expiration date (line 22). Notice the use of a callback to implement a reactive action (line 23).

The `Logout` function completes the implementation:

```
27 function logout() {
28    RconnState = "disconnected";
29    if( Rintv ) {
30       clearInterval( Rintv );
31       Rintv = false;
32    }
33 }
```

It merely changes the connection status (line 28) and clears the JavaScript timer if it is active.

This JavaScript source code looks relatively straightforward w.r.t our specification. Nevertheless, we make two observations. First, the global state variables `Rname`, `Rpasswd`, `RconnStatus`, `RenableLogin`, `Rintv`, and `Rconn` build hidden control dependencies between the different components. In practice, the number of such control variables grows with the complexity of the problem. For less trivial problems, and in particular when more and more features get added to the specification, state variable interactions can lead to a large number of implicit and pretty invisible global control states. Understanding the possible control flows and checking their consistency can become very complex. Second, the components are not isolated from each other. For instance, the `authenticate` component that requests for authentication and starts a session calls the `logout` component to execute some cleanup. This means that these components cannot be developed and tested independently, and that even local specification changes may involve modifying several components in a non-trivial way. These are the very situations where HipHop.js will show its full benefit, as we shall see in Section 3 when modifying our login design.

2.2 A HipHop.js Implementation

We now present our HipHop.js implementation, introducing the HipHop.js reactive constructions as we go and showing

how HipHop.js attacks the aforementioned programming problems. Let us first summarize HipHop.js's principles.

2.2.1 Principles of HipHop.js. HipHop.js's synchrony hypothesis [4, 5] is expressed as follows. A HipHop.js program executes conceptually instantaneous steps that are called *reactions* or *instants*. Within each instant, reactions execute HipHop.js statements in sequence or in parallel, letting them communicate using *instantaneously broadcast signals*. Execution and signal broadcasting instantaneity ensure that concurrency is *deterministic*, unlike asynchronous concurrency; error-prone manual synchronization is replaced by automatic and correct run-time scheduling, as decribed in Section 5 below.

Technically, HipHop.js separates two levels of languages: a synchronous control level in the language proper, based on specific temporal statements such as `abort`, `every`, `fork`/`par` etc, which will be illustrated in the sequel, and plain JavaScript data computations and test expressions used by these statements. External and internal communication is achieved by *signals*, which are divided into three classes: *input signals* set by the host JavaScript program, *output signals* returned to JavaScript when the reaction terminates, and *local signals* used internally.

Syntactically speaking, HipHop.js is a DSL for JavaScript, with a specific syntax. The switch from plain JavaScript to HipHop.js occurs either for data expressions enclosed in HipHop.js statements or in the `hop {...}` instantaneous statement. HipHop.js programs are organized in *modules*. A module declares its input and output signals in its *interface*, which can be declared in the module header or separately.

At each reaction, each signal broadcasts a unique *present / absent status* to all active temporal statements in its scope. A *pure signal* `S` carries no more information and its status is recomputed at each reaction, either as sent by the JavaScript environment for an input, or by the program for a local or output. Output and local signals are absent by default and set present in each reaction only by executing one or several instantaneous "`emit S()`" statements. The status of `S` at current and previous instants are accessed in the HipHop.js program's JavaScript expressions using the forms `S.now` and `S.pre`. In addition to the status, a *valued signal* has a unique JavaScript *value* at each instant, which is persistent over instants unlike for the status and can be initialized at signal declaration time. The value is specified by the environment for an input; for a local or output, it is updated by the current value of *exp* each time an "`emit S(`*exp*`)`" HipHop.js statement is executed. If several values are emitted in the same instant, they are combined by a specific function declared with the signal. The current value is accessed from HipHop.js's JavaScript expressions using `S.nowval` for current instant and `S.preval` for the previous instant. Broadcasting implies that all simultaneously active JavaScript data computations and tests enclosed in HipHop.js statements see the same signal information.

The HipHop.js-JavaScript toplevel interface is achieved by a JavaScript *reactive machine* that contains the HipHop.js compiled code and provides a simple communication API between the JavaScript environment and the HipHop.js code. Before each reaction, the status and values of input signals can be set by the host JavaScript program and passed to HipHop.js via the reactive machine. Those of output signals are returned to JavaScript via the reactive machine when the reaction terminates. Those of HipHop.js local signals remain local to the compiled JavaScript code.

For each reaction, the compiled JavaScript code atomically performs a complete microscheduling of all HipHop.js active sequential, concurrent, and preemption temporal statements, see Section 5. HipHop.js takes benefit of JavaScript atomic execution to avoid any run-time interference between reactive execution and the evolution of the asynchronous environment. This makes the theoretical synchronous semantics of HipHop.js automatically respected in practice, independently of any valid HipHop.js compiling technology. This brings a crucial advantage over traditional synchronous languages implementations, where atomicity is difficult for the user to manually guarantee.

☞ **Remark:** HipHop.js interface signals are different from JavaScript events; but, in practice, it is common for a reactive machine to directly bind input signals to JavaScript events and to generate JavaScript events from HipHop.js output signals. But this is by no way mandatory.

2.2.2 The `Main` HipHop.js Login Module. Our main module has valued inputs `name` and `passwd`, with strings as values, and pure inputs `login`/`logout` for the GUI buttons. Its outputs are `enableLogin` to control the visibility of the login button, `connState` to display a connection status message, and `time` to display the current connection time. It involves three submodules, using the `run` keyword to distinguish running a temporal module from calling a JavaScript function. `Identity` reads the GUI and enables the login button, `Authenticate` calls the authorization service, and `Session` manages an active session:

```
1  hiphop module Main(in name="", in passwd="", in login,
2                     in logout, out enableLogin,
3                     out connState="disconn",
4                     inout time=0, inout connected) {
5     fork {
6        run Identity(...);
7     } par {
8        every(login.now) {
9           run Authenticate(...);
10          if(connected.nowval) {
11             run Session(...);
12          } else {
13             emit connState("error");
14          }
```

```
15      }
16    }
```

In the interface declaration, '=' defines the persistent initial values of valued signals, overridden by the first reception for an input or by the first emission for any signal.

The `run` construct (line 6, 9, and 11) instantiates a submodule in place by inlining its code and binding its environment signals in the current lexical scope. The form "`run M(...)`" means that each interface signal is implicitly bound to a signal of the same name in the lexical environment. Manual binding of differently named signals can be done by "`S as S1`" if `S` is the interface signal. It will be used in Section 3.

Semantically speaking, `fork`/`par` (lines 5, 7) runs branches in parallel in the same signal status/value environment. Operationally, at each reaction, it picks one branch to execute until it terminates or is blocked on a test for the status or value of some yet unknown signal, then another branch, etc. Signal emission in one branch can resume other branches, and so on. This reaction-internal microscheduling is invisible to the user but always gives deterministic and semantically correct results. Deadlocks can occur but are always detected and reported at runtime, with a compiler warning if such a dynamic deadlock is possible, see Section 5.

The `every` construct (lines 8) implements a preemptive loop that checks for a condition, here the presence of `login`. An `every` loop starts its body when its condition is true, but kills it immediately to restart it anew whenever the condition is met again in some further instant. It is *strongly preemptive*: at any instant where the body is alive and the condition is true, the current instance of the body *does not execute*, only the new one does. The condition can be any JavaScript expression with the visible signals qualified by `now`, `pre`, `nowval`, or `preval`. JavaScript variables can also be declared and used provided they are not *shared*, *i.e.,* read in one parallel branch and written in another branch, which would break determinism; they will not be used here.

The expression `login.now` is true when `login` is received in the instant. Then, `Authenticate` is started, a previously active `Session` being first automatically killed. When `Authenticate` terminates, the value of `connected` is tested (line 10). If true, `Session` is started to run the new session until further login, logout, or time out. When `Session` terminates, the `every` statement simply waits for a new `login`.

2.2.3 The `Identity` Module.

The `Identity` module detects when a login becomes possible and signals that fact. It is defined as follows:

```
18 hiphop module Identity(in name, in passwd,
19                        out enableLogin) {
20   do {
21     emit enableLogin(
22       name.nowval.length>=2 && passwd.nowval.length>=2);
23   } every(name.now || passwd.now)
24 }
```

Here, `every` is at the end of the `do` statement: the `emit` statement is executed at module start time and terminates instantly; it is re-executed whenever at least one of the two inputs `name` and `passwd` is present; `enableLogin` is emitted with a Boolean value true if and only if `name` and `passwd` both have 2 or more characters, which updates the GUI as explained in Section 2.4.

2.2.4 The `Authenticate` Module.

The `Authenticate` module checks the validity of the identity at each click on login and emits the `connected` signal with value true if the connection succeeds and false otherwise:

```
25 hiphop module Authenticate(in name, in passwd,
26                            out connState, out connected) {
27   emit connState("connecting");
28   async connected {
29     authenticateSvc(name.nowval, passwd.nowval)
30       .post().then(v => this.notify(v));
31   }
32 }
```

First, the body emits a new connection state (line 27) that updates the GUI automatically, see Section 2.4. Second, it executes an `async` statement (line 28) enriched with a completion signal, here `connected`, and a plain JavaScript statement that is expected to *take time* in term of reactions, *i.e.*, not to complete during the current reaction. When started, `async` immediately triggers the JavaScript evaluation of its body, locally blocking its local HipHop.js control thread but not blocking other parallel branches to let the rest of this global HipHop.js reaction proceed. At line 29, the HipHop.js `async` body is a JavaScript promise that invokes the non-blocking remote web service `authenticateSvc`. When this promise completes, the expression "`this.notify(v)`" (line 30) provokes the broadcast towards the rest of the program of the `connected` output with `v`'s value, and `Authenticate` terminates.

Two major points must be noted w.r.t. the JavaScript previous code. First, `Main`'s `every` outer loop (line 8 of `Main`) instantly kills and restarts `Authenticate` when a new `login` is received. Thus, pending authentications are automatically discarded without needing the counter used in JavaScript. Second, there is no need to explicitly execute the logout cleanup operation when a new login occurs, unlike for JavaScript. As we shall see later, this operation is automatically triggered by "`every(login.now)...`" in `Main`.

☞ **Crucial note:** The HipHop.js `async` signal-enriched form is *the* element that enables reactive programs to harmoniously blend synchronous and asynchronous computations. It makes it possible to control external asynchronous computations within the semantically well-defined and deterministic world of reactive synchronous computations. It is also a major evolution of HipHop.js w.r.t. Esterel's `exec`, which only offers a very restricted and complicated link to C.

2.2.5 The `Session` and `Timer` Modules.

The `Session` module is in charge of running a session and handling disconnection:

```
33 hiphop module Session(connState, time, logout) {
34   emit connState("connected");
35   abort(logout.now || time.nowval > MAX_SESSION_TIME) {
36     run Timer(time);
37   }
38   emit connState("disconnected");
39 }
```

It starts a new session by emitting `connState` to update the GUI and by starting a timeout timer. Here, the `abort` construct executes its body until the condition is met: it aborts the timer and terminates when the session duration is over.

The `Timer` module is part of the standard HipHop.js library but, for completeness, we include it here since it stresses another essential feature of HipHop.js:

```
40 hiphop module Timer(time) {
41   async {
42     this.react({[time.signame]: this.sec = 0});
43     this.intv = setInterval(() =>
44       this.react({[time.signame]: ++this.sec}), 1000);
45   } kill {
46     clearInterval(this.intv);
47   }
48 }
```

Our `async` wrapping of the JavaScript `setTimeout` function has two important aspects. First, in the `async` body, `this` refers to the `async` itself, which in turn refers to the reactive machine. In the inner JavaScript call to `SetInterval`, the form `this.react(...)` tells JavaScript to queue a reactive machine reaction every second with input the current value of `time` in seconds. Since the `setInterval` JavaScript function never terminates, there is no need for an `async` completion signal such as `connected` for `Authenticate`. Second, this `async` will be automatically killed either by the `Session`'s `abort` statement when its timeout condition is met or by the super-enclosing "`every(login.now)`" of `Main` if a new login occurs. In both cases, the `kill` clause of `async` will automatically free the timer resource by calling the `clearInterval` JavaScript function. Generally speaking, no user of `Timer` needs to explicitly call this cleanup action when it kills the timer for any reason. This is a major modularity enhancement.

2.3 Comparing JavaScript and HipHop.js

We are now in position for an early comparison of both HipHop.js and JavaScript implementations.

Control. JavaScript uses global variables to represent the state of the application; note that the ES6 `async/await` extensions do not help with this respect. HipHop.js uses an explicit *control state* defined by the concurrent positions in the code where the control has stopped at the end of each instant. This enables HipHop.js modules to be developed and tested separately, which facilitates application validation and the building and reuse of library modules.

Preemption. The power of preemption clearly appears in the `Main` and `Timer` modules. In `Main`, a new login cancels all current activities. In `Timer`, cleanup occurs automatically without caring about why the module is killed. Symmetrically, modules that kill `Timer` do not need to care about cleanup. This factoring out process can be viewed as a concurrent generalization of finalization in algorithmic programming languages, but with a clear and deterministic semantics compatible with concurrency thanks to perfect synchrony.

In summary HipHop.js's explicitly temporal programming style leads to *behavioral modularity*, which is missing in JavaScript. The differences with JavaScript might seem mundane for such a simple application. But they will become much more visible with greater application complexity, as we shall see in Section 3.

2.4 Integration within the Web Page

We finish the login application by describing the interplay between HipHop.js, JavaScript, and HTML. We use Hop.js [35], a multitier extension of JavaScript that makes it possible to intertwine server-side and client-side code in the same source code, using the '~' mark to distinguish them. Hop.js also extends JavaScript syntax with that of HTML, whose tags simply become an alternate syntax for standard function calls. Hop.js also extends HTML with *react* nodes that update their content automatically when the expression they depends on need to be re-evaluated. Finally, Hop.js supports CommonJS modules on both server and client. Although HipHop.js and Hop.js are independent and address different problems, they have been designed to be complementary: Hop.js helps programming the asynchronous code deployment and communication between servers and clients, while HipHop.js helps programming synchronous patterns on both sides. Here is the server HipHop.js code.

```
1  service login() {
2    return <html>
3      <head>
4        <script src="hiphop" lang="hiphop"/>
5        <script src="./login.hh.js" lang="hiphop"/>
6        <script defer>
7          const M=require("./login.hh.js");
8        </script>
9      </head>
10     <input onkeyup=~{M.react({name: this.value})}/>
11     <input onkeyup=~{M.react({passwd: this.value})}/>
12     <button class=~{this.disabled=!M.enableLogin.nowval}
13             onclick=~{M.react({login: true})}>
14       login
15     </button>
16     <react>status=~{M.connState.nowval}</react>
17     <button class=~{M.connState.nowval}
18             onclick=~{M.react({logout: true})}>
19       logout
```

```
20       </button>
21       <div class=~{M.connState.nowval}>
22         time: <react>~{M.time.nowval}</react>
23       </div>
24     </html>
25 }
```

In the generated HTML page sent by the server's `login` service, the `require` call loads the client module `./login.hh.js` from the server and binds its exports to the client-side `M` variable, whose value is the Hop.js code of a HipHop.js reactive machine. To perform a synchronous reaction to a GUI user action, the application invokes `M`'s `react` method (lines 10, 11, 13, and 18), with arguments the emitted input signal names and optionally their value. For instance, line 10, `name` is emitted with the contents of the input box, and, line 18, the machine reacts to `logout` button click by emitting `logout` with value `true`. For HipHop.js output signals, their names are attributes of the reactive machines; thus their statuses and values can be directly accessed after a reaction. Finally, note that in this example, all the reactive execution takes place on the client-side: all the `react` calls are executed inside ~-expressions.

In the above code, the boundary between traditional HTML code and reactive HipHop.js code is kept clear. Both can be developed separately, for instance by GUI designers for the HTML part and by computer engineers for the reactive part. This was one of our main requirements when designing Hop.js and HipHop.js.

3 Login Panel 2.0

To better illustrate HipHop.js's temporal programming power, let us evolve our login panel specification: for version 2.0, after a fixed number of unsuccessful connection attempts, the whole system should freeze for a quarantine period.

In JavaScript, we need to add a new register to count the number of unsuccessful authentications, and to modify `authenticate` to increment or reset this register according to the result of `authenticateSvc`. We also need to explicitly ignore pending authentication replies and modify `nameKeypress` and `passwdKeypress` to disable the login button when in quarantine. Finally, we need a quarantine boolean state register and a JavaScript timer to reset it when the timeout expires. Almost all the initial implementation components need to be modified and re-tested to handle the quarantine. This is obviously a serious reengineering problem (left to the reader).

In HipHop.js we simply add to the unmodified body of `Main` V1 a new parallel `Freeze` module that listens to `Main`'s `connected` signal and emits in turn two signals, `freeze` after a fixed number of successive unsuccessful connections and `restart` when the quarantine is over:

```
48 hiphop module Freeze(var max, var attempts,
49                      sig, tmo, freeze, restart) {
50   do {
51     await count(attempts, sig.now);
52     emit freeze();
53     abort(tmo.nowval > max) {
54       run Timer(tmo as time);
55     }
56     emit restart();
57   } every(sig.now && sig.nowval);
58 }
```

The behavior reads trivially. The module body is an infinite loop reset by every reception of `sig` with value true, in our example by any valid connection. The body of this loop is a temporal sequence: waiting for `attempts` receptions of `sig`, which provokes immediate emission of `freeze`, then waiting for timeout to emit `restart`.

The new `MainV2` program is as follows:

```
59 hiphop module MainV2(tmo) implements ${Main.interface} {
60   signal freeze, restart;
61   fork {
62     loop {
63       weakabort(freeze.now) { run Main(...); }
64       emit connState("quarantine");
65       emit enableLogin(false);
66       await restart.now;
67       emit connState("disconnected");
68     }
69   } par {
70     run Freeze(max=5, attempts=3, sig as connected, ...)
71   }
72 }
```

The interface of `MainV2` is directly imported of that of `Main` using `implements`. The `loop` form implements an infinite loop (line 62) with instantaneous restart when the body terminates (the body is not allowed to terminate instantly when started). The `weakabort` form (line 63) implements a *weak abortion*. It is similar to `abort` but does execute its body at abortion time, while `abort` would prevent this last execution. Here, when `freeze` occurs, `Main` is immediately killed and will only be restarted at next `restart`. The `weakabort` construct is necessary because it is `Main` that sends `connected` to `Freeze`. Using `abort` would provoke a causality problem leading to microscheduling deadlocks: `Main` would emit `connected (false)` that would provoke `emit(freeze)`, which itself would prevent `Main` to execute. This deadlock would be detected and an error message generated by HipHop.js.

The outer loop restarts the `weakabort` statement that immediately restarts `Main` afresh. It is important to note that when `weakabort` starts it does not immediately check `freeze.now`. In terms of abortion, `abort` and `weakabort` are *delayed*, that is, check their condition only at instants strictly following their start. It would also be possible to do the check at start time by using the `immediate` keyword between `abort` and the condition.

On line 70, we have used new syntactic elements for running the `Freeze` module. First, in addition to signals, a module interface can declare variables. Here the `Freeze` module declares the `max` and `attempts` variables. When invoking the module we pass them `5` and `3`. Second, we use the syntax "`sig as connected`" to mean that the signal named `sig` in `Freeze`'s interface is bound to the signal `connected` of the environment.

In conclusion, with HipHop.js, we have been able to evolve our application by reusing the previous design unchanged, which also avoids lots ot retesting. Readability and smooth evolution are prominent advantages of HipHop.js compared to traditional algorithmic programming languages for implementing complex behavioral applications.

4 Two Real-Life Applications

HipHop.js has been designed to help programming applications that rely on complex temporal behaviors and that use modern execution environments. GUI usually implemented as web pages and gateway or server computing fall into this category. In this section, we present two such applications. The first one is still under development. The second is used in production.

4.1 Medical Prescriptions

Drug prescriptions are most often temporal, specifying when to take a given drug with side conditions about mistakes not to be made. Unfortunately, they are often quite fuzzy and the delivery of drugs is not always well-traced in hospitals, which generates quite a large number of accidents damageable to patients [23]. We plan to use HipHop.js to make prescriptions temporally rigorous and ensure their full traceability using web/smartphone applications, and possibly later develop smart IoT pillboxes with the very same HipHop.js code. We illustrate this here with the prescription of Lisinopril, which is a major hypertension drug. This example is tiny, but work continues on more complex ones (with Pr. Steven Belknap, Institute for Public Health and Medicine, Northwestern University, USA).

4.1.1 Lisinopril Prescription as a Specification.

Ignoring the dosage and treatment duration that are not relevant here, the official Lisinopril prescription looks trivial:

> *Take 1 tablet once daily with time-window 8PM to 11PM and time-wall 8 hours between doses.*

This looks simple and natural for a doctor but, for a computer scientist, the scheduling looks underspecified, even nonsensical: if a dose is taken each day between 8PM and 11PM, why add that two doses must be separated by more than 8 hours, which is redundant? The hidden reason is that harmful delivery mistakes are frequent in hospitals. Another potential mistake is clearly missing: what if no dose is given for a long time? In order to automatize the enforcement of the prescription, we need to turn it into a rigorous temporal behavior definition, with appropriate warnings for minor mistakes and error messages for major ones.

Here is a question-answer session we had with a doctor. Q: what if a tablet is taken outside the specified window? A: the 8-11 window is better, but no big deal provided that the 8h distance constraint is satisfied. Q: what if no tablet is taken for a long time? A: that's real bad, 34h should be the maximum. Q: would you like a computerized system to help preventing the 8h and 34h errors? A: certainly, provided it is simple to use by patients and heavily saturated nurses. Q: would it be useful to automatically log all events for later individual error analysis and, more generally, data analysis on a large number of patients? A: absolutely!

We propose a simple HipHop.js program driving a web/smartphone GUI. Our design choices are as follows:
1. A time display shows a minute-base clock with background green during the 8PM-11PM period and orange outside this period; another time display shows when the previous dose was taken; two buttons `Try` and `Confirm` control tablet delivery and confirmation; a text display shows errors and warnings (a beeper could be trivially added).
2. Doses are delivered and recorded by two successive presses: first on `Try` to check that the timing is valid, and then on `Confirm` to assert that the dose has been swallowed.
3. The `Try` and `Confirm` buttons are smart. First, they are inactive when pressing them should be disallowed, e.g., for 8 hours after the last `Confirm` for `Try`, or does not make sense: `Try` should become inactive once pressed and `Confirm` should be active only after this moment and only until pressed. Second, they embed a timer that raises an alarm (blinking red, beeping, etc.): for `Try`, if the previous dose was taken more than 30h ago, as a warning for approaching the 34h limit; for `Confirm`, if not pressed in reasonable time after `Try`.
4. All user and system actions are recorded in a log file with their date (easy and not detailed here).

Of course, there is still room for mistakes, for instance if an actually taken dose remains unconfirmed after a long time. The log should help analyzing such mistakes.

4.1.2 The HipHop.js Implementation.

Here is the HipHop.js program, with a button module and a main module. After the login example, these programs should read trivially ans stress once more the interplay between synchronous sequence, concurrency, and abortion.

```
hiphop module Button (var d, in Tick, in  B,
                      out Active, out Alert) {
 emit Active(true); emit Alert(false);
 abort (B.now) {
    await count(d, Tick.now);
    do { emit Alert(true); } every(Tick.now);
 }
 emit Alert(false); emit Active(false);
}

hiphop module Lisinopril extends Intf {
```

```
signal InDoseWindow;
fork {
  run Clock (...);
} par {
  loop {
    DoseOK: fork {
     run Button(d=TryDelay, Tick as Mn, B as Try,
                Active as TryActive, Alert as TryAlert);
     // Try received, deliver but warn if out of dose window
     emit DeliverDose();
     if (!InDoseWindow.nowval) {
       emit TryNotInWindowWarning ();
     }
     // phase 2 : wait for confirmation, keep alerting if late
     run Button(d=ConfDelay, Tick as Mn, B as Conf,
                Active as ConfActive, Alert as ConfAlert);
     // confirmation received
     emit RecordDose(Time.nowval);
     break DoseOK;   // end of phases 1 and 2
    } par {
     // in phases 1–2, error if too long wait since last dose
     await count(MaxDoseInterval-MinDoseInterval, Mn.now);
     sustain NoDoseSinceTooLongError();
    }
    // phase 3 : wait for min delay to allow Try again
    abort count(MinDoseInterval, Mn.now) {
     every(Try.now) { emit TryTooCloseError(); }
    }
  }
 }
}
```

This program introduces a new HipHop.js construct: the `DoseOK` label coupled with "`break OK`". This `break` instantly terminates the `fork/par` statement, weakly preempting the other parallel branch that would otherwise keep warning for no dose after its delay. That a `break` also instantaneously preempts concurrent activities makes perfect and deterministic sense in synchronous concurrent languages, which is not the case for asynchronous languages - when it exists.

Adding a `Reset` interface button would be trivial by enclosing Main's body in "`every(Reset.now) {...}`". We did not do it because of column width limitation.

4.2 Skini: Massively Interactive Music

Skini [30] is a web-based music composition and execution environment for live performances with broad audience participation through connected smartphones, smart watches, tablets, or PCs. Skini is designed to help the composer to find a balance between structured and interactive music. The music is built from *patterns* that are brief composed music elements, dynamically assembled in a continuous flow by real-time audience interaction. The patterns characterize the architecture of the music to be played. Their successive selections by the audience during the show creates a unique interpretation of the musical composition. Skini is used on a regular basis for performances, concerts [1], and music teaching [2]. Note that Skini's music is based on the dynamic modification of a precomposed structure, which contrasts with the local actions most interactive music systems focus on.

Skini extensively uses Hop.js and HipHop.js. Its server-multiple clients architecture is implemented using Hop.js. Its GUI, mostly executed on audience mobile phones, is implemented in Hop.js and HTML as for the login example of Section 2.4. Audience interaction is entirely programmed in HipHop.js since it requires extensive and complex event-based programming. All musical components, from the score to synthesizer control, are modeled as synchronously concurrent HipHop.js modules that interact though synchronous signals.

4.2.1 Music Scores.

The composer first creates a set of *music patterns* used for his piece, which are short music segment of 1 or 2 seconds. Patterns are accessible for selection to the audience only via *groups* and *tanks* that are activated or deactivated upon audience interactions. Patterns in an active group (resp. tank) can be selected multiple times (resp. only once) by the audience.

The composed *musical path* is the sequencing of group and tank activations and deactivations. For instance, the composer may choose to activate a group of patterns only after a fixed number of patterns of another group. Group activation/deactivation may also depend on constraints between groups, such as exclusion rules between groups that involve incompatible instruments or enforced group sequences to avoid too repetitive selections by the audience. Managing patterns, groups, and tanks for an interactive composition is obviously an orchestration problem. With Skini's the solution is to build a HipHop.js reactive program whose interactive execution drives standard digital audio workstation (daw) for sound generation.

4.2.2 Score Programming.

A HipHop.js score program controls the groups and tanks dynamically proposed to the audience during the performance. Selecting a pattern has two effects: first, its music is planned to be played; second, it impacts the future of the music as it may activate or deactivate other groups and tanks. The general HipHop.js program structure is as follows:

- Each group is implemented as a data structure controlled by two HipHop.js signals: an input signal sent by an auditor who selects one of its patterns when the group is active, with input value the selected pattern; an output signal telling Hop.js to flag the group as activated or deactivated.
- Tanks are implemented as arrays of one-pattern groups.

[1] https://www.cirm-manca.org/manca2017
[2] https://aide-aux-projets.sacem.fr/actualites/les-fabriques-a-musique

- Groups that play together are implemented as `fork/par` HipHop.js constructs.
- Sequences of groups are simply implemented as code sequences.
- Dependencies between groups and tanks are implemented using wait and preemption statements.

Here is an excerpt of a typical HipHop.js Skini score:

```
1 abort(seconds.nowval === 20) {
2    emit ActivateCellos(true);
3    await count(5, CellosIn.nowval);
4    run TrombonesTank();
5    fork {
6       run TrumpetsTank();
7    } par {
8       run HornsTank();
9    }
10 }
```

This fragment defines the sequencing of decisions. It will run for 20s. It first activates the `Cellos` group and waits for 5 cello patterns to be selected by the audience. Then the selected patterns are queued and played by the synthesizer under the control of Hop.js, HipHop.js being not needed for this simple operation. After 5 selections of cello patterns, the score enables `TromboneTank` that plays once each of its patterns, as dynamically selected by the audience. When the tank is exhausted, the same process is started for both TrumpetsTank and HornsTank, which play synchronously.

Using HipHop.js for such scores has two benefits. First the orchestration constructs are aligned with the musical constraints expressed by the composer with groups and tanks. In the above fragment the sequencing of lines 2 and 3 implements a construct such as "after the audience has asked for 5 cello patterns, offer to play trombones". Second, HipHop.js score programming suppresses implementation details that would be unavoidable with another language, making score composing more direct and flexible. It becomes easy to experiment variations, postpone the moment where trombones become active, add a new set of instruments at will, deactivate a group after the audience has adopted a particular behavior, etc. In other words, HipHop.js enables the composer to develop his score incrementally using a high-level try-and-error approach.

5 HipHop.js Implementation

When developing HipHop.js, our goal was to augment the expressiveness and power of JavaScript reactive programming by adding the Esterel concurrency and preemption statements. It was not to develop new compiling techniques for these primitives. Many ways to compile them into sequential code have been developed in the past with different simplicity / efficiency trade-offs [32]. Our HipHop.js to JavaScript compiler borrows from these studies. It works in three phases. Phase 1 is textual parsing and construction of a HipHop.js abstract syntax tree. Phase 2, the core one, expands all nested control structures into synchronously concurrent boolean operations [6]. The structure it builds, borrowed from hardware design, is called an *augmented boolean circuit*; see below. The same process has been used by Esterel compilers to generate both hardware circuits and software code in production environments. Its big advantage is to fully and faithfully implement Esterel's *constructive semantics* [6], which is also the reference semantics for HipHop.js. Phase 3 finally builds the run-time reactive machine code by generating its interface and then its reaction function, implemented as a linear-time software simulator of the circuit's electrical behavior.

We also provide an API to directly build abstract syntax trees from within JavaScript, which makes it possible to skip Phase 1 by constructing and compiling abstract HipHop.js code on the fly, this even on web client side; we plan to do this for future net-discovery based applications. Thus, the real core of the compiler consists only of phases 2 and 3; it is quite small, about 4.000 lines of JavaScript code, and the compiling time of a HipHop.js program is roughly proportional to its source code size.

5.1 From HipHop.js to Augmented Boolean Circuits

The augmented boolean circuit generated by Phase 2 is an extension of the classical notion of a *sequential circuit* in hardware design. It consists of a list of equations between *nets*, a hardware name for boolean variables. Input nets have no equation, and the other nets have a single equation that can be of two types. A *combinational equation* defines a net from other nets using a boolean expression; it can be augmented by a data expression corresponding to a JavaScript expression in the HipHop.js program, and by data dependencies to other augmented nets due to HipHop.js's statement sequencing. The calculation and assignment are instantaneous. A *register equation* implements a unit delay, as for `pre` in Lustre [21]. Its lhs net is assigned an initial value at first reaction; at each following reaction, the rsh value computed in this reaction is assigned to the lhs net only at the very end of the reaction, or equivalently at the start of the next reaction (at the next clock rising edge in hardware).

5.2 Run-Time Execution

Execution is linear in the number of net connections and data dependencies. It always terminates with an empty fifo if the generated circuit contains no *combinational cycle*, i.e, cycle involving only `and`, `or`, `not` expressions or data dependencies. Only a combinational cycle can block execution, by reaching a *synchronous deadlock*: think for instance of the equation "`X=not X`", which corresponds to "`if(!X.now){emit(X)}`" in HipHop.js: it means "emit `X` if you don't receive it". This is an obvious contradiction to the synchrony principle that states that a signal is received if and only if emitted; see [6] for an extensive discussion of synchronous causality. However,

some cycles that always lead to correct execution can be useful in making synchronous programs more symmetric or in reducing their size [27]. At runtime, correct cycles are correctly computed, but synchronous deadlocks cycles are always detected with an appropriate error message. This is a major advantage compared to deadlocks in asynchronous languages and systems, which block the system or part of it with no error detection and reporting.

Note that the fact that JavaScript executes this value propagation in an atomic way is absolutely critical for its correctness, since it forbids any change to the inputs during the execution; any input status or value change in the middle of the execution would violate synchrony making it semantically wrong. JavaScript's atomicity is definitely a bless.

Our evaluation scheme has strong relation with both denotational semantics and electrical propagation of voltages in digital circuits. In the pure case (no data expressions and tests), our forward propagation system algorithm computes the least fixpoint semantics of the equations interpreted in Scott's ternary logic on $\{\bot, 0, 1\}$, where $\bot$ means undefined. It also exactly mimics the stabilization of voltages in circuits during a clock cycle, see [27].

5.3 Memory Footprints and Time Performance

The generated circuit size is most often linear in HipHop.js source code size. But quadratic expansion can occur in special cases, due to a subtle phenomenon called reincarnation of statements and signals by loops. Reincarnation makes the same source statement execute several times in the same instant but in different environments. Although semantically quite subtle, reincarnation is technically easy to handle at compile time [6, 37]. It is fully supported by HipHop.js.

For HipHop.js execution, each net is implemented as a JavaScript object containing 12 to 15 properties: the array of other nets it is connected to, its undefined or boolean state, the number of nets it still depends on because of control or data dependencies, the JavaScript data expression if needed, a reference to its reactive machine, and some debug information. For large programs, nodes are on average connected to two other nets. The number of bytes per net depends on the underlying JavaScript implementation. With Hop.js on a 64bit platform, any JavaScript object is 32 bytes plus 8 bytes per property, and an array has 6 extra words. That is each node is on average in between 192 and 216 bytes. The Lisinopril application presented in Section 4.1 is compiled into 399 nets, using about 86KB of memory. Skini music scores are much bigger programs (see Section 4.2). For instance, a typical classical music score[3] can compiled into up to 10,000 nets, which occupy about 2.1MB of memory.

Reaction execution time is roughly linear in circuit size. Obviously, this raises no performance problem for small applications such as Lisinopril but it might become problematic for very large applications. For instance Skini scores are quite large programs and subject to stringent time constraints because their executions must be kept in sync with the external synthesizers that drive the music. A score pulse is about 100-200 beats per minute, which means that Skini reactions must complete within at most 300ms. We have measured that even for the largest available score the HipHop.js reaction time never exceeds 15ms, so the current speed is clearly good enough to follow the pulses.

Our current circuit simulation balances simplicity of compilation and execution with decent speed. If more speed is needed in the future, more size- and speed-efficient techniques for implementing synchronous reactive languages are available [32] and will be implemented in HipHop.js.

6 Related Work

HipHop.js is a JavaScript-based direct descendant of the Esterel synchronous language, which was designed in the 80's for static embedded systems such as conventional real-time controllers. HipHop.js adapts Esterel's ideas and style to the dynamic world of web programming, where JavaScript is prominent. From Esterel its borrows the programming style, the core semantics and some implementation techniques. Its major advantage its that the integration with JavaScript is much easier and tighter than Esterel's loose integration with C, especially for the linking of asynchronous and synchronous activities that was limited and cumbersome in Esterel. This is essential for mixed asynchronous / synchronous applications. HipHop.js subsumes former preliminary attempts [7, 38] that used an API with no concrete syntax and a rudimentary connection between JavaScript and the synchronous parts. Also, albeit not discussed here for lack of space, HipHop.js is dynamic at source-code level: it allows the user to partially reconfigure the program between two synchronous reactions while keeping the control state consistent. This is necessary to cope with the web programming model where programs can be sent by servers through the network after client configuration discovery for instance; synchrony is crucial for making this process safe.

In the object-oriented community, reactive systems are traditionally implemented using the observer pattern [19]. The Scala.React technical report [26] and the IoT security survey [8] show the weakness of this practice with an extensive list of arguments. The main one is the limitation to implement a state machine with global data structures and side effects, which makes complex observer-based designs difficult to implement correctly, maintain, and modify as their composability is weak. These are the very reasons that have motivated the design of HipHop.js to be based on synchronous concurrency, which is composable by construction.

There are two kinds of synchronous concurrent languages. On the one hand, those geared to the systematic use of preemptive structures that control the life and death of activities,

[3] https://soundcloud.com/user-651713160

mainly Esterel and the synchronous graphical programming-oriented offsprings of Harel's Statecharts [2, 10, 22, 39]. On the other hand, those oriented to stream-based programming, such as Lustre [21] and Signal [20]. We have shown that preemption is quite fundamental for the applications we aim at, which is why we chose Esterel as a basis. But fancy stream-based programming is not yet available in HipHop.js, which is still a weakness (SCADE 6 [10] combines both aspects, but in a simpler purely synchronous framework).

More recently, SCL [40] is a very interesting and promising extension of Esterel main notions for C where signals are replaced by standard variables, with an even richer microscheduling. It would be a good start for a web-related work similar to ours but with single language integration in view, provided it does not restrict too much the rest of the host language.

Web-oriented dataflow reactive languages have been widely used to program GUIs as they can be used to relieve the programmer from imperatively updating the interface. Functional Reactive Programming, initially proposed in the context of lazy languages [16] has been adapted to imperative languages such as Scheme [11, 12], F# [31] and JavaScript. Flapjax [28] was the first of this kind for the Web; it has inspired other language designs such as Elm [14] and UrWeb [9]. It has also probably inspired many industrial frameworks such as JavaFx [13] Adobe Flex [36], React.js [17], or Vue.js [41]. Hop.js `react` DOM nodes find their inspiration in these studies. But, as for UrWeb `dyn` tag, Hop.js's tags are more focused than all these frameworks as they only serve one purpose: the update of precisely identified DOM nodes. These constructs are not really relevant to the control-based orchestration issues we want to address.

LuaGravity [34] and later Scala.React [26] both propose embedding synchronous reactive programming in a general purpose language. They propose to blend reactive and sequential programming using *reactors*, which are similar to our HipHop.js reactive machine. But neither LuaGravity nor Scala.React enforce a clear boundary between the reactive and sequential components. This dramatically impacts the programming style, the implementation, and the tooling.

Reference [24] evaluates the impact of reactive programming on web applications. Its authors compare different programming models, in particular JavaScript promises and JavaScript-based data-flow programming as found in Flapjax or React.js. Its authors mostly focus on the programming of the *Model-View-Controller* pattern. We think that the problem of implementing reactive systems is more general. For instance, none of the examples considered in their study required preemption as used here. We think that preemptive operations are absolutely fundamental, for instance when processing parallel queries to handle the first answer and abort the other queries. The Orc programming language [25] was designed specially with this objective in mind. It supports high-level operators for combining, forking, and synchronizing streams. These operations are complementary with HipHop.js that yet supports no notion of temporal streams, although LuaGravity gives some direction for encoding them on top of our reactive machine. Integrating the notion of temporal stream into HipHop.js might be a direction for future work.

Managing the states of web applications is the explicit objective of various popular industrial frameworks. For instance, Flux [18] and Redux [1] are two JavaScript frameworks created to better organize the handling of the global state. Flux's architecture is unidirectional: user actions are broadcast by a dispatcher to a set of local stores that each hold part of the global state and that themselves can send back change signals to update the application's view. HipHop.js implicit signal broadcasting makes this pattern easy to implement using local signals to store the state components and output signals to realize the updates. Redux has no dispatcher but a single store and *reducers* that are functions in charge of parts of the state. The main reducer and its sub-reducers, if concerned by the action, return a new state from the unmodified current state. The new global state is obtained by composing these results. This pattern is also easy to implement in HipHop.js using signals to store state components, with `S.pre` and `S.preval` for the original state and `S` and `S.val` for the new state status and values. Comparing such patterns implemented in standard languages with dedicated domain-specific languages is a question to be permanently debated.

Microsoft's *durable functions* [29] is an orchestration library available for C# and JavaScript. It supports a subset of the HipHop.js core constructs: parallel and sequential composition, synchronization, pause, and asynchrony / synchrony blending, but only rudimentary preemptive structures.

7 Conclusion

We have shown how to unify three basic programming paradigms: transformational, interactive, and reactive. For the reactive part, we have introduced the reader to HipHop.js, which incorporates Esterel's synchronous reactive programming model and semantics into Hop.js that itself facilitates programming the interactive part. JavaScript takes care of transformational aspects, Hop.js of interactive aspects, and HipHop.js of reactive aspects, in a tightly integrated way. Thanks to synchronous technology, HipHop.js concurrent constructs are compiled into equivalent plain JavaScript sequential code that can be atomically executed either on servers or on clients.

We have illustrated the HipHop.js programming style by three examples to show that programming and reusing temporal behaviors is facilitated by synchronous sequencing, concurrency, signalling, and preemption, the main addition of HipHop.js w.r.t. JavaScript and Hop.js.

☞ **url:** http://hop.inria.fr/hiphop

References

[1] D. Abramov and A. Clark. 2018. Redux. https://redux.js.org

[2] C. André. 1996. Representation and Analysis of Reactive Behaviors: a Synchronous Approach. In *Proc CESA'96, IEEE-SMC, Lille, France.*

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. 1996. *Concurrent Programming in ERLANG* (second edition ed.). Prentice Hall.

[4] A. Benveniste and Berry. G. 1991. The synchronous approach for reactive and real-time systems. *Proc. IEEE* 79, 9 (1991), 1270–1282.

[5] G. Berry. 2000. The Foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte (Eds.). MIT Press.

[6] G. Berry. 2002. *The Constructive Semantics of Pure Esterel.* Draft book, current version 3. http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf

[7] G. Berry and M. Serrano. 2014. Hop and HipHop: Multitier Web Orchestration. In *Proc. ICDCIT 2014, Bubhaneswar.* Springer-Verlag Lecture Notes 8337.

[8] Z. B. Celik et al. 2018. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. arXiv:1809.06962 https://arxiv.org/pdf/1809.06962.pdf

[9] A. Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 153–165. https://doi.org/10.1145/2676726.2677004

[10] J-L. Colaço, B. Pagano, and M. Pouzet. 2017. SCADE 6: A formal language for embedded critical software development. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE).*

[11] G. Cooper and S. Krishnamurthi. 2004. FrTime: Functional Reactive Programming in PLT Scheme. Tech Report CS-03-20 (April 2004).

[12] G. Cooper and S. Krishnamurthi. 2006. Embedding dynamic dataflow in a call-by-value language. In *In European Symposium on Programming.* 294–308.

[13] Oracle Corporation. 2010. JavaFX. http://javafx.com

[14] E. Czaplicki and S. Chong. 2013. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Notices.*

[15] ECMA. 2015. ECMAScript Language Specification. http://www.ecma-international.org/ecma-262/6.0/

[16] C. Elliott and P. Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming.* http://conal.net/papers/icfp97/

[17] Facebook Inc. 2019. A JavaScript library for building user interfaces. https://reactjs.org/

[18] Facebook Inc. 2019. Flux: In Depth Overview. https://facebook.github.io/flux/docs/overview.html

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[20] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with Signal. 1991. Programming real-time applications with Signal. *Proc. IEEE* 79, 9 (10 1991), 1321 – 1336.

[21] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous dataflow programming language Lustre. 79, 9 (1991).

[22] D. Harel. 1987. Statecharts: a Visual Approach to Complex Systems. *Science of Computer Programming* 8 (1987), 231–274.

[23] J. T. James. 2013. A new, evidence-based estimate of patient harms associated with hospital care. *Journal of Patient Safety* 9, 3 (September 2013).

[24] K. Kambona et al. 2013. An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications. In *Workshop on Dynamic Languages and Applications.*

[25] K. Kitchin and *et al.* 2009. The Orc Programming Language. In *Formal Techniques for Distributed Systems.*

[26] I. Maier, T. Rompf, and M. Odersky. 2012. Deprecating the Observer Pattern.

[27] M. Mendler, T. Shiple, and G. Berry. 2012. Constructive Boolean Circuits and the Exactness of Timed Ternary Simulation. *Formal Methods in System Design* 40, 3 (2012), 283–329.

[28] L. Meyerovich et al. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09).* ACM, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

[29] MicroSoft. 2019. Durable Functions patterns and technical concepts. https://docs.microsoft.com/en-us/azure/azure-functions/durable

[30] B. Petit and M. Serrano. 2019. Composing and Executing Interactive Music Using the HipHop.JS Language. Porto Alegre, Brazil.

[31] T. Petricek and D. Syme. 2010. Collecting Hollywood's garbage: Avoiding space-leaks in composite events. In *Proceedings of International Symposium on Memory Management.*

[32] D. Potop-Butucaru, S. A Edwards, and G. Berry. 2007. *Compiling Esterel.* Springer Science & Business Media.

[33] J. Reppy. 1999. *Concurrent Programming in ML.* Cambridge University Press.

[34] F. Sant'Anna and R. Ierusalimschy. 2009. LuaGravity, a Reactive Language Based on Implicit Invocation. In *Proceedings of SBLP'09.* 89–102.

[35] M. Serrano and V. Prunet. 2016. A Glimpse of Hopjs. In *21th* ACM SIGPLAN *Int'l Conference on Functional Programming (ICFP).* Nara, Japan, 188–200. https://doi.org/10.1145/2951913.2951916

[36] Adobe Systems. 2010. Flex quick start: Handling data. http://www.adobe.com/devnet/flex/quickstart/using_data_binding

[37] O. Tardieu and R. de Simone. 2005. Loops in Esterel. *ACM Transactions on Embedded Computing Systems (TECS)* 4, 4 (2005), 708–750.

[38] C. Vidal, G. Berry, and M. Serrano. 2018. Hiphop.js: a Language to Orchestrate web Applications *(extended abstract).* In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France.* 2193–2195. https://doi.org/10.1145/3167132.3167440

[39] R von Hanxleden et al. 2014. SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).* 372–383.

[40] R. Von Hanxleden et al. 2014. Sequentially Constructive Concurrency - A Conservative Extension of the Synchronous Model of Computation. *ACM Transactions on Embedded Computing Systems (TECS), Special Issue on Applications of Concurrency to Systems Design* 13:144 (2014), 1–26.

[41] E. You. 2019. The Progressive JavaScript Framework. https://vuejs.org/