Hiphop.js: a language to orchestrate web applications
Colin Vidal, Gérard Berry, Manuel Serrano

To cite this version:

HAL Id: hal-01937252
https://hal.archives-ouvertes.fr/hal-01937252
Submitted on 28 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Hiphop.js: a language to orchestrate web applications

Colin Vidal
Université Nice Côte d’Azur
Inria Sophia Méditerranée
colin.vidal@inria.fr

Gérard Berry
Collège de France
gerard.berry@college-de-france.fr

Manuel Serrano
Université Nice Côte d’Azur
Inria Sophia Méditerranée
manuel.serrano@inria.fr

CCS CONCEPTS
• Software and its engineering → Domain specific languages;
Orchestration languages;

KEYWORDS
Web Programming, Reactive Programming, Orchestration

ACM Reference Format:

1 INTRODUCTION

We are interested in web orchestration, which is the problem of appropriately handling the asynchronous events appearing in program executions. It is known be one of the major difficulties of web programming (see callback hell [9, 11]).

High-level approaches to orchestrate web applications have been developed in academia or industry. The Functional Reactive Programming (FRP) concept [4] has been carried up to the web by Flapjax [5, 10], which adopts a dataflow programming style: when a variable is modified, any expression that references it is implicitly reevaluated. Others techniques targeting GUI updates [1, 6, 8, 12] consist in associating a state with a set of graphical elements and automatically updating the graphics on changes. In JavaScript Promises and async/await constructions [7] make it possible to chain asynchronous actions in a specific sequential order. Working at a more abstract level, these solutions avoid using callbacks.

Our goal is to go further with yet another solution based on a new language called Hiphop.js. It is a JavaScript extension of Esterel [2] based on three reactive control mechanisms: explicit concurrency, synchronisation using synchronous signals, which makes the handling of concurrent issues much easier, and preemption, i.e. the explicit cancellation of an ongoing orchestration subactivity. Hiphop.js suitably extends the core Esterel notions to deal with web paradigms such as application structure dynamicity. Hiphop.js follows a previous Scheme-based prototype [3], but takes different approaches w.r.t. the interaction with the host language.

This paper gives an informal introduction to Hiphop.js through a small example that compares the programming of a simple event-aware program in both JavaScript and Hiphop.js.

2 SPECIFICATION OF THE EXAMPLE

We study the timer application of the 7 GUIs project 1 where the timer’s duration is represented by a slider, the current elapsed time is displayed both by a gauge and as a number, and a Reset button resets the timer.

When running the timer the gauge and numeric field are constantly updated until the duration is reached. From then on, setting a bigger duration immediately restarts the timer from the current elapsed time. At any time, pressing Reset resets the elapsed time to 0 and restarts the timer.

3 IMPLEMENTATION

We compare the JavaScript and Hiphop.js timer implementations, the latter introducing the Hiphop.js programming style and constructions.

The plain HTML GUI is common to JavaScript and Hiphop.js:

```
<div>Elapsed time: <meter id="meter"/></div>
<div><span id="elapsed">s</span></div>
Duration: <input id="range" type="range" max="100" onchange="setD(this.value)"/>
<button onclick="resetE()">Reset</button>
```

3.1 JavaScript implementation

The JavaScript code is as follows:

```javascript
var D=0, E=0, id=-1;
window.onload = function () {
  $('meter').value=E; $('meter').max=D;
  $('elapsed').innerHTML=E; $('range').value=D;
}

function tick () {
  if (E<D) {
    E +=0.1; $('meter').value=E;
    $('elapsed').innerHTML=E;
  }
  if (E==D) { clearTimeout (id); id = -1; }
}

function startIntervalIfNeeded () {
  if (id == -1 && E<D) id=setInterval(tick, 100);
}

function resetE () {
  E = 0; $('meter').value=E; clearTimeout (id); id = -1;
  startIntervalIfNeeded();
}

function setD (value) {
  D = value; $('meter').max=D;
}
```

1 https://github.com/eugenkiss/7guis/wiki/time
3.2 Hiphop.js implementation

Let us first ignore the Reset button with a BasicTimer:

```javascript
MODULE BasicTimer {
    IN duration(0);
    OUT elapsed;
    EMIT elapsed(0);
    LOOP {
        IF (VAL(elapsed) < VAL(duration)) {
            RUN(TimeoutMod(100));
            EMIT elapsed(PREV(VAL(elapsed) + 0.1));
        } ELSE {
            PAUSE;
        }
    }
}
```

All Hiphop.js keywords are capitalized. `MODULE` defines a module, here the whole Hiphop.js program. `IN` and `OUT` declare the program interface. A Hiphop.js program is executed as a succession of atomic reactions, each handling input signals and possibly generating output signals.

Contrary to JavaScript scattered callbacks, in Hiphop.js there is a unique temporal code, which is based on classical sequencing and looping, signal emission, and temporal statements. Each statement has a temporal duration in term of reactions. `EMIT` emits a signal and terminates immediately, i.e. passes control in sequence in the very same reaction. `LOOP` immediately starts its body and restarts it immediately when it terminates. `PAUSE` pauses for one reaction.

Let us now explain how `BasicTimer` executes. At initial reaction, `LOOP` starts its body that emits the `elapsed` signal with value 0. The `IF` test is then evaluated. If the value of `elapsed` is less than that of `duration`, the `TimeoutMod` module is ran. As presented in Section 7 this will terminate the reaction, and after 1/100 will resume with the `elapsed` emission. Otherwise, the reaction terminates.

Adding Reset is achieved by another temporal statement called `LOOPEACH` placed around the unmodified `BasicTimer` module:

```javascript
MODULE Timer {
    IN duration(0) , reset ;
    OUT elapsed ;
    LOOPEACH(NOW(reset)) {
        RUN(BasicTimer);
    }
}
```

`LOOPEACH` immediately starts the `BasicTimer`. From then on, whenever Reset is pressed, this timer is killed whichever state it was in, and a new timer is immediately restarted afresh. The advantage of temporal programming to modularly describe behavior becomes obvious: no internal state manipulation of `BasicTimer` needs to be performed by the user.

4 EXTENSION OF THE SPECIFICATION

The differences between Hiphop.js and JavaScript are more visible as the orchestration problem gets more complex. Let us add a new functionality to the previous specification: the timer can be suspended when it is running. In that case, the elapsed time is no longer incremented until the timer is resumed. The duration slider and Reset button remain active during suspension. In the GUI, we add a Suspend button that toggles the timer between the normal and suspended mode. The Suspend button turns to orange during suspension.

5 EXTENDED IMPLEMENTATION

The GUI is extended with a Suspend button:

```html
<button id="susp" onclick="susp()">Suspd</button>
```

5.1 New JavaScript implementation

Events triggered by the Suspend button are handled by:

```javascript
function susp() {
    if (isSusp) {
        isSusp = false;
        $("susp").style.backgroundColor = "transparent";
        startIntervalIfNeeded();
    } else {
        isSusp = true;
        $("susp").style.backgroundColor = "orange";
        clearTimeout(id); id = -1;
    }
}
```

A new global variable `isSusp` is used to keep the current state of the suspension. The previous code needs adaptations:

```javascript
function resetE() {
    if (isSusp) {
        isSusp = false;
        $("susp").style.backgroundColor = "transparent";
    } else {
        isSusp = true;
        $("susp").style.backgroundColor = "orange";
        clearTimeout(id); id = -1;
        startIntervalIfNeeded();
    }
}
```

```javascript
function setD(value) {
    D = value; $("meter").max = D;
    if (!isSusp) startIntervalIfNeeded();
}
```

The orchestration becomes more complex to understand; the whole program has been modified in a non-local way by adding `isSusp`, patching existing functions, and adding another function.

5.2 New Hiphop.js implementation

The new Hiphop.js SuspendableTimer is as follows:

```javascript
MODULE SuspendableTimer {
    IN duration(0), reset, suspend;
    OUT elapsed, suspendColor;
    LOOPEACH(NOW(reset)) {
        FORK {
            SUSPEND TOGGLE(NOW(suspend)) {
            RUN(BasicTimer);
        }
    PAR {
        EMIT suspendColor("transparent");
        LOOP {
            AWAIT(NOW(suspend));
        }
    }
}
```
The code is now made of two parallel arms for controlling the JavaScript timer and for coloring the Suspend button. Since both arms are included in the LOOP EACH, they will be both simultaneously killed and restarted when reset occurs.

Between resets, both arms work in lockstep, i.e. conceptually synchronously. The suspend input signal is broadcast. The first arm automatically suspends and resumes the timer as specified, while propagating its termination. The second arm toggles the button color.

As for the first Hiphop.js implementation, the code makes the temporal behavior explicit, the syntactic nesting of temporal construction explicitly specifying their lifetimes and priorities. States are in the code, not in the data, and the BasicTimer code can be directly reused without modification.

6 LINKING HIPHOP.JS AND JAVASCRIPT

This section explains the link between Hiphop.js and JavaScript. Let us call stm the reactive machine compiled from SuspendableTimer and describe its API. When Reset button is pressed, the GUI calls the resetE JavaScript function. In Hiphop.js, we make this function send the reset signal and trigger a reaction:

```javascript
function resetE () {
    stm.input("reset"); stm.react(); }
```

The input signals duration and suspend are respectively handled by functions setD and susp in the same way.

Conversely, to transform signal output by Hiphop.js into JavaScript actions, we associate Hiphop.js event listeners with output signals. For instance, the following code updates the gauge and numeric field:

```javascript
stm.addEventListener("elapsed", evt => {
    \$\("meter\"\).value = evt.signalValue;
    \$\("elapsed\"\).innerHTML = evt.signalValue; });
```

The other outputs are handled in the same way.

7 CONTROLLING ASYNCHRONOUS ACTIONS

The EXEC Hiphop.js statement is used to launch and control external actions whose execution spans several Hiphop.js reactions: a XMLHttpRequest, a setTimeout call, etc.

Here is the source code of the parametric setTimeout Mod timer submodule, parametrized by a number of milliseconds:

```javascript
function TimeoutMod (nms) {
    return MODULE {
        let id;
        EXEC id = setTimeout(DONEREACT, nms)
        ONKILL clearTimeout(id)
        ONSUSP clearTimeout(id)
        ONRES id = setTimeout(DONEREACT, nms);
        }
    }
}
```

The Hiphop.js "EXEC start-expr" statement executes the JavaScript expression start-expr. Then, it pauses at each reaction, terminating only when the action has completed. When EXEC starts, a JavaScript function is automatically created, it is referred to as DONEREACT in Hiphop.js. Its call will trigger both the termination of EXEC and a new reaction. An EXEC statement also specifies three optional side-effecting expressions: the ONKILL expression is automatically evaluated when the EXEC statement is killed, for example here by the "LOOP EACH(NOW(reset))" enclosing statement. The ONSUSP expression is automatically evaluated when EXEC gets suspended, and the ONRES statement when it gets resumed.

8 CONCLUSION

Callback-based orchestration of web applications in JavaScript is known to be quite difficult and hazardous. Better solutions have already been studied in academic and industrial contexts. In our opinion, they are not yet fully satisfactory since they are either too invasive (dataflow programming style), too specific (targeting only the GUI updates), or still limited (promises and async/await).

The Hiphop.js language extends JavaScript with a richer orchestration solution. Our approach has been to port Esterel concepts and techniques to the web, basing orchestration of web applications on dedicated temporal statements.

In our opinion, Hiphop.js has two important virtues. First, it deals with events orchestration in a very behavioral and modular way, promoting straight code reuse over deep modification of existing code. Second, being smoothly embedded into JavaScript and interfaced with standard code, it does not impose a drastic technical change to users.

REFERENCES