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

To cite this version:
Colin Vidal, Gérard Berry, Manuel Serrano. Hiphop.js: a language to orchestrate web applications. SAC: Symposium on Applied Computing, Apr 2018, Pau, France. 10.1145/3167132.3167440. hal-01937252

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.
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 to 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:

```html
<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;
}

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

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

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

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

1https://github.com/eugenkiss/7guis/wiki/time

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SAC 2018, April 9–13, 2018, Pau, France
© 2018 Copyright held by the owner/author(s).
https://doi.org/10.1145/3167132.3167440

2193
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(PREVAL(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 terms 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 run. 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, suspend;
  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="susps()">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 {
    E = 0; $({"meter").value = E;
    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 \texttt{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 theirs lifetimes and priorities. States are in the code, not in the data, and the \texttt{BasicTimer} code can be directly reused without modification.

6 \hspace{1em} LINKING HIPHOP.JS AND JAVASCRIPT

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

\begin{verbatim}
function resetE () {
    stm.input("reset"); stm.react();
}
\end{verbatim}

The input signals duration and suspend are respectively handled by functions \texttt{setD} and \texttt{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:

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

The other outputs are handled in the same way.

7 \hspace{1em} CONTROLLING ASYNCHRONOUS ACTIONS

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

Here is the source code of the parametric \texttt{TimeoutMod} timer submodule, parametrized by a number of milliseconds:

\begin{verbatim}
function TimeoutMod (nms) {
    return MODULE {
        let id;
        EXEC id = setTimeout(DONEREACT, nms)
        ONKILL clearTimeout(id)
        ONSUSP clearTimeout(id)
        ONRES id = setTimeout(DONEREACT, nms);
    }
}
\end{verbatim}

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

8 \hspace{1em} 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 \texttt{async/await}).

The Hiphop.js language extends JavaScript with a reacher 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


