HIPHOP: A Synchronous Reactive Extension for HOP

Gérard Berry
Inria Sophia Méditerranée
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France
Gerard.Berry@inria.fr

Cyprien Nicolas
Inria Sophia Méditerranée
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France
Cyprien.Nicolas@inria.fr

Manuel Serrano
Inria Sophia Méditerranée
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France
Manuel.Serrano@inria.fr

Abstract

HOP is a SCHEME-based language and system to build rich multi-tier web applications. We present HIPHOP, a new language layer within HOP dedicated to request and event orchestration. HIPHOP follows the synchronous reactive model of the Esterel and ReactiveC languages, originally developed for embedded systems programming. It is based on synchronous concurrency and preemption primitives, which are known to be key components for the modular design of complex temporal behaviors. Although the language is concurrent, the generated code is purely sequential and thread-free; HIPHOP is translated to HOP for the server side and to straight JAVASCRIPT for the client side. With a music playing example, we show how to modularly buid non-trivial orchestration code with HIPHOP,

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages

General Terms Design, Languages

Keywords JavaScript, Web Programming, Functional Programming, Reactive Programming, Synchronous Programming

1. Introduction

HOP [9] is a SCHEME-based programming language and system aimed at simplifying web programming. A HOP program embeds server code and client code in a single program. The HOP compiler automatically splits the compound code into server code run by the HOP runtime and client code compiled into JavaScript. HOP automatically manages data transmission and event communication between the server and client. It can also handle applications involving multiple servers and clients.

This paper introduces the HIPHOP extension[1] that adds a new dimension to HOP: the sophisticated handling of events based on *synchronous reactive programming* [1], with event-handling and concurrency programming primitives inspired by the Esterel [3] and ReactiveC [5] languages. Synchronous reactive programming has become classical in the circuit design and real-time programming areas, where concurrency is ubiquitous and event handling is a crucial concern. It has many industrial implementations in embedded systems such as airplane, train, or plant control, as well as complex HMIs such as airplane cockpits [4].

We show that synchronous programming techniques are also well adapted to web programming, and especially to complex requests handling. We demonstrate that embedding synchronous concurrency and event-handling primitives into HOP makes web programming easier and has the potential of greatly simplifying the development of web applications.

Section 2 presents basic examples, the way they are programmed in JavaScript, and the way they are

[1] Work partially supported by the French ANR agency, grant ANR-08-EMER-010

simplified by HIPHOP. Section 3 details the reactive core of HIPHOP. Section 4 explains how the new statements are incorporated in HOP to build HIPHOP and how client-side synchronous event handling and concurrency are translated into JavaScript. Section 5 briefly presents a real-life example. Section 6 gives an overview of the HIPHOP implementation. We discuss related work in Section 7 and conclude in Section 8.

2. A First Example

JavaScript event handling is based on scripts and event listeners, which are functions registered to handle asynchronous occurrences of events generated by the GUI, server, and APIs. Scripts and event listeners can be used to mimic a kind of implicitly parallel execution within the JavaScript basically sequential language. However, as known for long in the real-time community, such callback-based code is subject to all possible kinds of interference and much more difficult to develop, verify and maintain than code based on explicit concurrency and explicit preemption primitives.

Our first example will be about coordinating basic requests. Here the traditional XMLHttpRequest (henceforth XHR) JavaScript example found in the Mozilla documentation "*Using XMLHttpRequest*".

```
function xhr(url) {
   function transferComplete(evt) {
      // do something on completion
   }
   function transferFailed(evt) {
      // signal an error
   }
   var req = new XMLHttpRequest()
   req.addEventListener("load", transferComplete, false)
   req.addEventListener("error", transferFailed, false)
   req.open("GET", url, true)
   req.send(null)
}
xhr("http://www.mozilla.org");
```

This basic example opens a HTTP connection to the Mozilla web server that fetches the main HTML page. The translation to HOP is direct:

```
(define (xhr url)
   (define (transfer-complete evt)
      ;; do something on completion
      )
   (define (transfer-failed evt)
      ;; signal an error
      )
   (let ((req (new XMLHttpRequest)))
      (add-event-listener! req "load" transfer-complete)
      (add-event-listener! req "error" transfer-failed)
      (req.open "GET" url #t)
      (req.send)))

(xhr "http://www.mozilla.org")
```

The translation to our new HIPHOP reactive style is more complex since it cuts work into two parts: a reactive HIPHOP program and the HOP linking part. The linking part is very similar to what we had before except that it creates a HIPHOP reactive machine and links the events to it.

```
(define (xhr url)
   (define (transfer-complete evt)
      ;; do something on completion
      )
   (define (transfer-failed evt)
      ;; signal an error
      )
   (let ((M (make-hiphop-machine (xhr& url))))
      (add-event-listener! M "failed" transfer-failed)
      (add-event-listener! M "complete" transfer-complete)))

(xhr "http://www.mozilla.org")
```

The HIPHOP reactive machine is driven by a HIPHOP program, built here by calling the following HOP function with a URL as argument:

```
(define (xhr& url)
   (let ((req (new XMLHttpRequest)))
      (hiphop&
         input: "load" "error"
         output: "complete" "failed"
         (atom& (let ((M (the-machine)))
                   (connect-event-listener! req "load" M)
                   (connect-event-listener! req "error" M)
                   (req.open "GET" url #t)
                   (req.send)))
         (until& "error"
            (seq&
               (await& "load")
               (emit& "complete" (val& "load")))
            (emit& "failed" (val& "error"))))))

(define (connect-event-listener! req ev M #!optional opt)
   (add-event-listener! req ev
      (lambda (e)
         (hiphop-input-and-react! M (or opt ev) e))))
```

All HIPHOP constructs end with the "&" character. The `hiphop&` function defines the interface and body of the reactive code, which is thought of as a state machine. The initial `atom&` form simply executes its HOP argument. Here, it connects actual events to the symbolic inputs of the machine and initiates the request. The rest of the HIPHOP code is of a more temporal nature. The `await&` form waits for the "`load`" reactive event and terminates. The first `emit&` form emits the "`complete`" reactive event bound to "`transfer-complete`" in the linking code. The `seq&` form tells that `await&` and `emit&` act in sequence. The `until&` form acts as a watchdog: if "`error`" occurs during the execution of the sequence, that sequence is immediately canceled and "failed" is emitted. Values are transmitted using the val& form.

So far, the HIPHOP version looks more complex than the plain HOP version for no good reason. The advantage will become clear when we modularly extend the program. Suppose first we want to synchronize two independent XHRs, running both of them in parallel and executing the transfer-complete listener only when both have completed. We simply create a new program that concurrently composes two xhr& HIPHOP programs and places them into an error watchdog:

```
(define (xhr2& url1 url2)
   (hiphop&
      input: "load1" "error1" "complete1"
             "load2" "error2" "complete2"
      output: "complete" "failed"
      (until& (or& "error1" "error2")
         (seq&
            (par&
               (run& (xhr& url1)
                  input: "load" "load1"
                  input: "error" "error1"
                  input: "complete" "complete1")
               (run& (xhr& url2)
                  input: "load" "load2"
                  input: "error" "error2"
                  input: "complete" "complete2"))
            (emit& "complete"))
         (emit& "failed"))))

(define (xhr2 url1 url2)
   (define M (make-hiphop-machine (xhr2& url1 url2)))
   (add-event-listener! M "failed" transfer-failed)
   (add-event-listener! M "complete" transfer-complete))

(xhr2 "http://www.mozilla.org" "http://hop.inria.fr")
```

The run& form instantiates a HIPHOP program, binding the proper reactive events that are now renamed into "load1" / "load2", etc. The par& form runs the two xhr& instances in parallel. Its semantics is to terminate when both have terminated. Therefore it is sufficient to emit "complete" in seq& sequence after the par& parallel to signal global completion. The until& form captures errors in both xhr& instances. If there is no such error, it has no effect ; as soon as one error occurs, it aborts both xhr& instances and executes the (emit& "failed") statement that signals global failure.

Imagine we now want to control our parallel XHR by offering the user an abort button and limiting the request durations by a timeout. We create a third HIPHOP program that controls the behavior of xhr2&. It waits for xhr2& to complete or fail until the user clicks the abort button or the timeout expires:

```
(define (xhr3& url1 url2)
   (hiphop&
      input: "load1" "error1" "load2" "error2" "abort"
      output: "complete" "failed" "user-abort" "timeout"
      (until& (timer& 1000)
         (until& "abort"
            (run& (xhr2& url1 url2))
            (emit& "user-abort"))
         (emit& "timeout"))))

(define (xhr3 url1 url2 id)
   (define M (make-hiphop-machine (xhr3& url1 url2)))
   (add-event-listener! M "failed" transfer-failed)
   (add-event-listener! M "complete" transfer-complete)
   (add-event-listener! M "user-abort" user-abort)
   (add-event-listener! M "timeout" user-abort)
   (add-event-listener! (dom-get-element-by-id id)
      "click" (lambda (e) (hiphop-input! M "abort"))))

(<HTML>
   (<BUTTON> id: "abort-button"
      "click me to abort downloading"))
```

In summary, HIPHOP is based on a split between event linking and event reaction specification, the latter based on explicit sequential, concurrent, and temporal programming primitives. We exemplified this with a typical client program. Of course, HIPHOP can also be used on the server-side, for example to synchronize clients together or complex requests chains using other servers. This will not be studied in this paper.

3. The HIPHOP language

Technically speaking, a HIPHOP form should be seen in two ways. First, it is a HOP form that builds a HIPHOP abstract syntax tree when evaluated, which implies that we can dynamically build HIPHOP programs from within HOP; we shall do that in Section 5. Second, the intention is that of concurrent and temporal code. In this section, we describe what HIPHOP temporal programming means.

3.1 HIPHOP programs and machines

The HIPHOP language specifies a *reactive program* that can be instantiated in HOP to build an executable *reactive machine*. A HIPHOP reactive program specifies a list of abstract input events, a list of abstract output events, and a reactive code. Here is a standard example in the reactive community, which can be seen as a reduced version of our xhr3&:

```
(define abro&
   (hiphop& input: "A" "B" "R" output: "O"
      (loop&
         (until& "R"
            (par&
               (await& "A")
               (await& "B"))
            (emit& "O")))))
```

A HIPHOP program is embedded into a machine M in the following way:

```
(define M (make-hiphop-machine <hiphop-program>))
```

Note that the same program can be embedded in several different machines. Thus, make-hiphop-machine acts as new in object-oriented systems.

3.2 Events and reactions

At HIPHOP level, events are abstract objects known only by their name (string) and optional value (arbitrary HOP data). In the previous examples, we have shown how to link HOP actual events to HIPHOP abstract events using HOP event listeners. Besides input and output events, a HIPHOP program can also declare *local events*, which behave in the same way from the HipHop point of view but need no HOP interface. Local events help synchronizing the concurrent parts of the HIPHOP program when dealing with complex reactive behavior. In the sequel, "[...]" mark optional arguments, "[...]*" possibly empty lists of arguments, and "[...]+" the same non-empty.

Input events are sent to a machine M by the HOP statement:

```
(hiphop-input! M "A" [<hop>])
```

The optional <hop> value will be the value associated with the abstract event "A" in the HIPHOP reactive code. This only sets the input, without triggering execution of M.

The machine is explicitly driven from HOP. It reacts only when called using the following HOP form:

```
(hiphop-react! M)
```

Then, the input environment to which M reacts is composed of all the input events that have been sent to the machine from HOP since the previous reaction (or since machine build at first reaction). All these events are perceived as conceptually simultaneous by M, which reacts by building output events according to its HIPHOP code execution. The execution of the reaction should be seen as atomic: outputs computed in a reaction should not influence the inputs for this reaction. This is to avoid interference between input event registering and reaction.

When the machine is called to react is left to the user. It is often useful to call the reaction function each time an input even is sent, but many other possibilities make sense, such as making the machine react at periodic times. To trigger a reaction in as soon as an input occurs, one can simply write:

```
(hiphop-input-and-react! M "A" [<hop>])
```

In each reaction, an event has exactly one *status* chosen between *present* and *absent*. An input event is present if and only if it was sent by HOP before the reaction (and after the previous reaction if any). An output or local event is present if and only if it is emitted by the HIPHOP program in the current reaction. The status is recomputed at each reaction.

An event also has an optional *value*, which is received from the environment using hiphop-input! for an input event or internally emitted together with the status for an output or local event. Unlike the status, the value is memorized between reactions: the value of an event in a reaction remains the value it had in the previous reaction if the event is not received (for an input) or emitted (for an output or local) in the reaction.

The value is either a single HOP value if there is only one emitter for the event during the reaction or the multiset of all emitted values if there are several emitters. The use of multisets instead of lists is important to guaranty determinism because multiset operations are associative: the multiset does not depend on the internal emission order in the generated code execution.

In each reaction, all events and event values are broadcast to all concurrent statements of the program, which all see the same statuses and values. Unlike asynchronous concurrent programs, concurrent HIPHOP programs are deterministic.

3.3 The reactive code

The reactive code is based on deterministic sequencing, concurrency, and temporal statements inspired from Esterel [3] and ReactiveC [5]. Their intuitive semantics is described below; see [2] for a complete presentation of synchronous programming, including formal semantics.

HIPHOP differs from classical languages by the temporal character of its execution: current control positions are memorized from one reaction to the next. Consider the following sequence:

```
(seq&
   (await& "A")
   (await& "B")
   (emit& "O"))
```

At first reaction, control stops on (await& "A"). It stays there at each subsequent reaction until the first reaction where "A" is present. In this reaction, control immediately moves to (await& "B") and stays there until the next reaction where "B" is present. During this reaction, and without further delay, it outputs "O" and terminates.

We say that a statement that starts and terminates in the same reaction is *instantaneous* or *immediate*; this is the case for emit&. Otherwise, we say that the statement *pauses*, waiting for the next reaction, and we call it a *delay statement*; this is the case for await&. Things that happen in the same reaction are called *simultaneous*. This is of course a conceptual notion in terms of abstract reactions, not a physical one.

Here is the list of HIPHOP values and statements:

```
<hiphop> →
    (hiphop& [input: [<string>]+] [output: [<string>]+]
         [<hiphop-stmt>]+)

<hiphop-stmt> → (nothing&)
  | (emit& string [<hop>])
  | (sustain& string [<hop>])
  | (atom& <hop>)
  | (if& <event-test> <hiphop-stmt> [<hiphop-stmt>])
  | (if& <hiphop-val> <hiphop-stmt> [<hiphop-stmt>])
  | (pause&)
  | (seq& [<hiphop-stmt>]+)
  | (par& [<hiphop-stmt>]+)
  | (loop& [<hiphop-stmt>]+)
  | (await& [immediate: <bool>] <event-test>)
  | (until& [immediate: <bool>] <event-test>
       <hiphop-stmt> [<hiphop-stmt>])
  | (local& ([<string>]+) [<hiphop-stmt>]+)
  | (let& ([(id <hiphop-val>)]+) [<hiphop-stmt>]+)
  | (run& <hiphop> [input: <string> <string>]*)

<event-test> → <string>
  | (or& <event-test>+)
  | (and& <event-test>+)
  | (not& <event-test>)

<hiphop-val> → (val& <string>)
  | <hop>
```

A hiphop& program is specified by an input event list, an output event list, and a reactive statement list (implicitly in seq& sequence).

An event test is a Boolean expression about events presence in the reaction. For instance, in a reaction, (and& "A" "B") is true if both "A" and "B" are present. The val& form returns the value attached to the mentioned event.

The nothing& statement does nothing and terminates instantaneously. It is the HIPHOP no-op.

The emit& statement emits the event named string, possibly with value that of the second argument. It is instantaneous.

The sustain& statement keeps emitting the event at each reaction. It can be defined in HOP using the pause& statement defined below:

```
(define (sustain& event #!optional val)
   (loop&
      (emit& event val)
      (pause&)))
```

The atom& statement calls HOP to executes its hop argument; it is instantaneous, which means that it should be a reasonably simple HOP expression in practice. The let& statement below makes it possible to pass HIPHOP values to the atom body, see below.

The if& statement evaluates its test. If the result is true, it immediately executes its second HIPHOP argument; otherwise, it immediately executes its third argument. Notice that these arguments can be arbitrary instantaneous or delayed HIPHOP statements. Termination of the if& statement is instantaneously triggered by termination of the selected branch. As before, values can be passed to a HOP test using let&.

The pause& statement delays execution by one reaction. When executed, it pauses for the reaction and terminates at the next reaction.

The seq& statement executes its arguments in order: the first HIPHOP statement starts immediately when the sequence starts; when it terminates, be it immediately or in a delayed way, the second argument is immediately started, etc. For instance, (seq& (emit& "A") (emit& "B")) immediately emits A and B, which are seen as simultaneous within the reaction, while (seq& (emit& "A") (pause&) (emit& "B")) emits "A" and "B" in two successive reactions.

The loop& statement is a loop-forever, equivalent to the infinite repetition of its argument statements, which are themselves implicitly evaluated in sequence. For instance, (loop& (pause&) (emit& "A")) waits for the next reaction and then keeps emitting "A" at each reaction. A loop can only be exited by using the until& statement below.

The par& statement starts its arguments concurrently and terminates at the reaction where the last of them terminates. Therefore, (par& (await& "A") (await "B")) terminates when both "A" and "B" have been received. Remember that all arms of a par& statement see all statuses and values of all (visible) events in exactly the same way.

The await& statement is a delay statement that waits for its argument to become true. By default, it pauses in the reaction where it is started; then, at each reaction, it evaluates the condition and terminates if the result is true; otherwise, it pauses again. If `:immediate #t` is specified, the `await&` statement terminates immediately if its condition is true at start time. It is then a shorthand for: `(if& event-test (nothing&) (await& event-test))`.

For the `until&` statement, call the second argument the *body* and the third argument the *handler* (assumed to be `nothing&` if syntactically absent). The body is run until it terminates or the event test becomes true. More precisely, at each reaction, the body is run for the reaction. If it terminates, so does the `until&` statement. Otherwise, the event test is evaluated; if its value is true, then the body is killed whichever state it was in and control passes to the handler. The event test is not performed at first reaction for the default form; it is performed when `:immediate #t` is specified. Note that the handler is not executed if the body terminates normally and the event test is false.

The `local&` statement declares local events in the first argument list. Their scope is the body, which is the implicitly `seq&` sequential list of the HIPHOP arguments. Such local events behave exactly as input and output events, except that their scope is limited to the `local&` body. They are not visible from HOP. A `local&` statement terminates when its body does.

The `let&` statement makes it possible for HOP forms called in `emit&`, `atom&`, and `if&` statements to access the values of HIPHOP events. The syntax is similar to that of the HOP `let`. Bindings are of the form `(v (val& "S"))` to transfer the value of a event, or `(v hop)` to perform any kind of HOP binding. A `let&` statement terminates when its implicitly `seq&` sequential body does.

The `run&` statement instantiates another HIPHOP program in-place, according to a list of abstract event bindings that act as textual substitutions for the body. It is the basis of modular HIPHOP programming. The optional argument list specifies how events are bound between the caller and callee, by enumerating pairs of the form `:input caller-string callee-string`. It is useless to specify an identity pair such as `:input "A" "A"`, which is implicit. A binding of the form `:input "" "A"` leaves the "`A`" callee event unbound and transforms it into a local event for the callee's body. The `run&` statement terminates when its body does. It may be killed by an `until&` statement as for any other statement.

3.4 Future additions

We plan to add the Esterel `suspend&` statement that suspends execution of its body for one reaction when its event-test is true, as well as Esterel *tasks* [2]. Tasks are external asynchronously concurrent computation entities whose execution is assumed to be non-instantaneous. For instance, a task could be "close the roller blinds" or "download the 5th Beethoven symphony". Their execution can be tightly controlled by `until&` and `suspend&` statements.

4. HOP Integration

In this section we present the interface between HOP and HIPHOP, *i.e.*, the HIPHOP machine API.

As explained in Section 3.1, building an HIPHOP machine `M` from HIPHOP program `P` is realized by:

```
(define M (make-hiphop-machine P))
```

Notice that HIPHOP programs are HOP values. Therefore we can use the full power of HOP to build them, depending, for instance, on size parameters. This will be done in Section 5.

Sending an input to `M` with an optional HOP value is done using the `(hiphop-input! M "A" [hop])` form. Although this form can appear anywhere in HOP programs, it is good practice to associate it with event listeners as done in our examples.

Outputs are systematically handled by HOP event listeners attached to the machine. These are registered with the following forms:

```
(add-event-listener! M string function)
```

where `string` is the output event name and `function` is a one-argument function. The argument is the event descriptor that contains the event name and optional value. The function is called when the output is emitted by the HIPHOP program. It is possible to register several events listeners for the same output event, which are invoked as for JavaScript.

Running a reaction of the machine `M` is done by calling `(hiphop-react! M)`. This call blocks until the reaction completes. It can itself involve the evaluation of HOP forms in `emit&`, `atom&`, and `if&` statements, see Section 3.3.

Of course, the major difficulty is to generate the HOP and JavaScript codes that performs the reaction according to the HIPHOP specification. We shall give no details here, see for instance [8] for Esterel compilation techniques.

A very important point is that the generated code is purely sequential and it can be fully and faithfully implemented in straight JavaScript. Although HIPHOP code can be highly concurrent in its own way, its concurrency is purely logical and does not lead to run-time threads. Program behavior remains deterministic w.r.t. input sequences, unlike with all forms of asynchronous concurrency-based programming techniques.

Notice also that HIPHOP machines can also communicate with each other in HOP, possibly using standard asynchronous web protocols. For instance, a HIPHOP machine can drive a complex UI on the client, another HIPHOP machine can drive complex server mashup activities involving a variety of other asynchronous servers, and both machines can communicate asynchronously using HOP server/client communication primitives. This is akin to the Globally Asynchronous Locally Synchronous (GALS) model that has become standard in other Computer Science fields such that Systems on Chips (SoC) design.

5. A real life example

We want to take organized benefit of the numerous sources that provide musical contents with the help of some specialized search sites that reply questions about musical composers, songs, etc., by lists of URLs where music can be found. The first step is to ask a question to the search site and to gather the URLs (not done here). Then, we try playing the music contents in sequence. Some musical downloads may fail, other may come fast, slow, etc. Our choice is to play music in a greedy way. We first play according to the fastest response. Then, we continue according to which download is ready next, selecting one if many are. The whole application completes when all the available music pieces have been played. Of course, some downloads may fail.

Our architecture is picture in Figure 1. It consists of a HIPHOP program concurrently composed of an orchestrator HIPHOP subprogram `orch&`, a set of requesters `req&`, one per URL returned by the search engine, and a music player controller `player&`. At program creation time, the requesters are built and indexed by the URL they are responsible for. The set of URLs needs not be known by `orch&`.

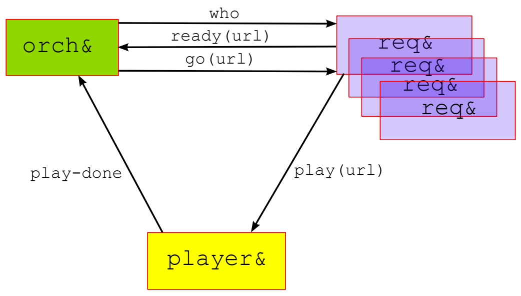

Figure 1. The multiple URLs audio player

The subprograms chat with each other as follows:

- When the program starts or each time a music play is over, `orch&` keeps broadcasting the event "`who`" to all the requesters.
- When ready, each requester replies by an event "`ready`" with value its URL. A requester that receives a download error simply terminates.
- When `orch&` receives "`ready`", by definition of HIPHOP simultaneous event emissions, the value "`ready`" is the multiset composed of all the URLs of the ready requesters. At that time, `orch&` selects one of the requesters and replies with the event "`go`" with value the selected URL.
- Still in the same reaction, the selected requester sends the "`play`" event to `player&` and terminates. The other ready requesters discard "`go`" and continue waiting for "`who`". The non-ready or terminated requesters ignore "`go`" altogether.
- When receiving "`play`" with the URL, the `player&` starts playing the music, until completion or error. Then, it sends "`play-done`" to `orch&`, which loops and looks for another ready requester.

In the main program called `urlplayer&`, the orchestrator, requesters, and player are put in parallel. To handle global termination, we introduce a local event "`all-done`" and an `until&` statement to terminate the whole behavior, killing the orchestrator and player. The "`all-done`" event is emitted when all requesters and the currently played music have terminated.

Here is the code of the main program `urlplayer&`. Notice the use of a HOP `map` on the URL list to build the requesters:

```
(define (urlplayer& urls)
   (let ((loads (rename "load-" urls))
         (errors (rename "error-" urls)))
      (hiphop& input: loads errors "ended"
         (local& ("who" "ready" "go" "play" "play-done"
                  "reqs-done")
            (until& (and& "all-done" "play-done")
               (par&
                  (seq&
                     (all-reqs& urls loads errors)
                     (sustain& "reqs-done"))
                  (run& orch&)
                  (run& player&)))))))

(define (all-reqs& urls loads errors)
   (par&
      (map (lambda (u l e)
              (run& (req& u)
                 input: "load" l
                 input: "error" e))
         urls loads errors)))

(define (rename prefix urls)
   (map (lambda (u) (string-append prefix u)) urls))
```

The code of orch& is fairly trivial. Notice the use of sustain& to keep broadcasting “who” until some requester is ready:

```
(define orch&
   (hiphop&
      input: "ready" "play-done"
      output: "who"
      (loop&
         (until& "ready"
            (sustain& "who"))
         (let& ((url-list (val& "ready"))
                (url-go (apply select url-list)))
            (emit& "go" url-go)
            (await& "play-done")))))
```

The req& program first runs the xhr& subprogram of Section 2 to start downloading. It terminates if a download error occurs. If the download is complete, req& waits for “who” and immediately replies by “ready” with value its URL. In the same reaction, orch& replies with “go” and the URL it has selected as “go” value. If the selected URL is the req&’s URL, this req& is selected and immediately ships its URL to the player& and provokes its own termination by emitting “done”, which is caught by the enclosing until&. Otherwise, req& waits for the next “who”:

```
(define (req& url)
   (hiphop&
      input: "who" "go" "load" "error"
      output: "ready" "play"
      (local& ("complete" "failed" "done")
         (run& (xhr& url))
         (if& "complete"
            (until& "done"
               (loop&
                  (await& immediate: #t "who")
                  (emit& "ready" url)
                  (let& ((go (value& "go")))
                     (if& (eq? go url)
                        (seq&
                           (emit& "play" url)
                           (emit& "done"))
                        (pause&))))))))
```

The player& just plays the music when requested and emits “play-done” when done:

```
(define player&
   (hiphop&
      input: "play" "ended"
      output: "play-done"
      (loop&
         (await& immediate: #t "play")
         (let& ((url (val& "play")))
            (atom& (set! audio.src url)))
         (await& "ended")
         (emit& "play-done"))))
```

To finish the whole HOP program, we need to link the HIPHOP code to a HOP master. This is almost trivial since most of the work was already done in xhr&. We only need to define the main function and perform an audio “ended” event connection for the actual HTML5 player:

```
(define (urlplayer urls)
  (<HTML>
    (<AUDIO> onended:
       ~(hiphop-input-and-react! M "ended" event))
    ~(let ((M (make-hiphop-machine (urlplayer& urls))))
       (hiphop-react! M))))
```

6. Implementation

In this section, we explain how HIPHOP programs are compiled and executed on the server and client sides. Figure 2 sketches the associated architecture.

6.1 HIPHOP to HOP compiling

HIPHOP programs are parsed by the server’s HOP system using the hiphopc function, which translates HIPHOP programs into HOP values that represent their ASTs (abstract syntax trees). For server-side execution, the ASTs are directly taken as input by the server’s HOP compiler and HIPHOP runtime. For client-side execution, the ASTs are processed further

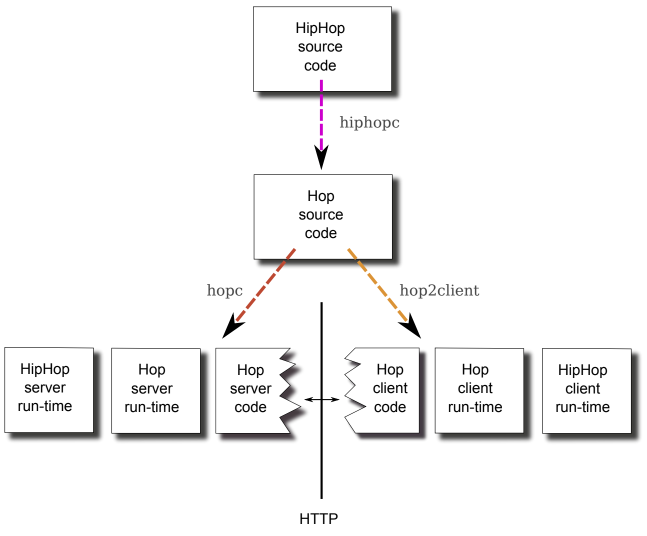

Figure 2. HIPHOP architecture

by the hop2client function that generate client-side code.

Notice that the hiphopc and hop2client functions reside on the server. Therefore, if the client wants to compile HIPHOP source code (for instance, interactively entered by the user), the client sends this code to the server, which processes it by hiphopc and hop2client and sends back the result.

6.2 The HIPHOP runtime

The HIPHOP runtime manages reactive machines by defining the functions mentioned in Sections 3.2 and 4: make-hiphop-machine and all the functions whose names start by hiphop-.

The HIPHOP runtime is written as a HOP module. Thus, any HOP program running on the server can import the HIPHOP module and use its API code to interface reactive machines. To run reactive machines on the client-side, the HIPHOP runtime is compiled to client code by calling the hop2client HOP function described above.

Communication between client and server reactive code requires no particular addition to HOP.

Causality cycles The heart of the HIPHOP runtime is an interpreter that executes the reactive instructions specified in the HIPHOP ASTs. This interpreter is based on the Constructive Semantics described in [8]. In the synchronous languages field, it is well-known that synchronous concurrency can provoke causality cycles, i.e., internal synchronous deadlocks. Such cycles are detected at run-time by the HIPHOP interpreter; they raise an exception and abort execution instead of stupidly deadlocking as would asynchronous threads do. In the future, we will add a static analysis pass to detect the absence of causality cycles before compiling, as for Esterel.

Timers In addition to the basic reactive primitives, HIPHOP provides the user with a timer& function, which we used in section 2 within the definition of xhr3&. Executing (timer& delay) generates an internally named event that occurs after (at least) delay milliseconds and make the HIPHOP code react to it.

On the server side, the implementation of timer is direct in HOP. On the client side, it uses the JAVASCRIPT timers, which are asynchronous entities that call back some event handler code at the specified time. Client HOP uses JAVASCRIPT timers to defines the after HOP function, which takes as arguments a delay and an argumentless function to be called when the timer expires. This mechanism is used to define timer:

```
(define (timer& delay)
   (lambda (current-machine)
      (let ((event (format "Timer-~a" (gensym))))
         (after delay
            (lambda ()
               (hiphop-input! current-machine event)))
         event)))
```

7. Related work

Our first example, showed in Section 2, is about coordinating requests. The Orc language [6] addresses this issue by proposing a process calculus composed of three basic combinators: symmetric composition, dynamic parallel-for loop, and pruning. Our par& form is akin to Orc's symmetric composition, while until& is a way to preempt computations similar to pruning but more general. The temporal algebra of HIPHOP is richer than that of Orc. However HIPHOP does not offer the flexibility of the Orc $f > x > g$ operator that dynamically creates parallel executions and connects data streams, which is fundamental for large-scale data processing. We do not know yet how to incorporate such dataflow primitives in HIPHOP.

Flapjax [7] provides a unified framework for programming with events on the client-side. The authors highlight three principles that should ease programming mashup applications: *event-driven reactivity*, *consistency*, and *uniformity* when treating events. We obey the same principles: the HIPHOP machine only needs to react upon new inputs (e.g. external events);

consistency is guaranteed by the atomicity of a reaction (i.e. output events cannot change the status of input events within a reaction); HIPHOP events are all handled using the same set of primitives. Furthermore use the `addEventListener` primitive to register output events handler on an HIPHOP machine, syntactically sticking to the DOM standard's primitive.

Orc and Flapjax are dataflow languages. Data channels are implicit in Orc: the combinators implicitly build and connect channels between expressions. Flapjax uses explicit event streams in conjunction with flow behaviors to manipulate data. Evaluation is asynchronous, and values are propagated at any time. Flapjax choose to use the topological order to prioritize computation and avoid *glitches*. Because of the reactive semantics, a safe scheduling is computed at compile-time, there is no need for priorities, and there are no glitches. Event and data causality is respected at runtime by construction.

8. Conclusion

We have presented HIPHOP a new way to orchestrate activities within HOP. HIPHOP deals with abstract events linked to actual web, UI, or API events by trivial HOP linking code. The HIPHOP reactive statements are imported from the Esterel [3] and ReactiveC [5] languages, which were created in the 80's for programming embedded systems. They are based on temporal sequentiality, concurrency, and preemption. They have a well-understood and well-published formal semantics, not repeated here. Integrating these statements in HOP gives new possibilities in two directions: first, bringing the power of reactive programming into HOP-based web programming; second, because of the reflexive character of HOP that can build its own programs as data structures, making it possible to dynamically build and execute complex HIPHOP reactive code in function of the problem to solve and to the current environment of a client or server. Of course, much work remains to be done to implement HIPHOP in a really efficient way, to develop bigger web applications with it, and to incorporate orchestration mechanisms available elsewhere but not yet in HIPHOP.

References

[1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270 –1282, sep 1991.

[2] G. Berry. The Esterel language primer, version v5_91. Technical report, Ecole des mines de Paris and Inria, June 2000. http://www-sop.inria.fr/members/Gerard.Berry/Papers/primer.zip.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. de Simone. Esterel: a formal method applied to avionic software development. *Science of Computer Programming*, 36(1):5–25, 2000.

[5] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience*, 21 (4):401–428, 1991.

[6] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In C. Baier and H. Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006. ISBN 978-3-540-37376-6.

[7] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.

[8] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[9] M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, Portland, Oregon, USA, Oct. 2006.