Hop and HipHop : Multitier Web Orchestration

Gérard Berry[1] and Manuel Serrano[2]

[1] Collège de France,11 place Marcelin Berthelot, 75231 Paris Cedex 05, France,
Gerard.Berry@college-de-france.fr
[2] Inria Sophia Méditerranée, 2004 route des Lucioles, 06902 Sophia Antipolis, France,
Manuel.Serrano@inria.fr

Abstract. Rich applications merge classical computing, client-server concurrency, web-based interfaces, and the complex time- and event-based reactive programming found in embedded systems. To handle them, we extend the Hop web programming platform by HipHop, a domain-specific language dedicated to event-based process orchestration. Borrowing the synchronous reactive model of Esterel, HipHop is based on synchronous concurrency and preemption primitives that are known to be key components for the modular design of complex reactive behaviors. HipHop departs from Esterel by its ability to handle the dynamicity of Web applications, thanks to the reflexivity of Hop. Using a music player example, we show how to modularly build a non-trivial Hop application using HipHop orchestration code.

1 Introduction

Our aim is to help programming rich applications driven by computers, smartphones or tablets; since they interact with various external services and devices, such applications require orchestration techniques that merge classical computing, client-server concurrency, web-based interfaces, and event-based programming. To achieve this, we extend the Hop multitier web programming platform [8] by the new HipHop domain specific language (DSL), which is based on the synchronous language Esterel [1]. HipHop orchestrates and synchronizes internal and external activities according to timers, events generated by the network, GUIs, sensors and devices, or internally computed conditions.

Like Esterel, Hiphop is a concurrent language based on the perfect synchrony hypothesis: a HipHop program repeatedly reacts in conceptual zero-delay to input events by generating output events; synchronization and communication between parallel statements is also performed in conceptual zero-delay. Perfect synchrony makes concurrent programs deterministic and deadlock-free, the only non-determinism left being that of the application environment. Its implementation is cycle-based, execution consisting of repeated atomic cycles "read inputs / compute reaction / generate outputs" in coroutine with the main Hop code. Concurrency is compiled away by static or dynamic sequential scheduling of code fragments. Cyclic execution atomicity avoids interference between computation and input-output, which is the usual source of unexpected non-determinism and synchronization problems for classical event-handler based programming.

While Esterel is limited to static applications, HipHop is designed for dynamicity. Its implementation on top of Hop makes it possible to dynamically build and run orchestration programs at any time using Hop's reflexivity facilities. It even makes it possible to modify a HipHop program between two execution cycles (not detailed here). It also simplifies the language by importing Hop's data definition facilities, expressions, modular structure, and higher-order programming features. It relies on the Web asynchronous concurrency and messaging already supported by Hop.

Section 2 briefly presents the Hop language. Section 3 describes HipHop and its relation with Hop. Section 4 presents a music application. Section 5 briefly overviews related work. We conclude in Section 6.

2 Hop

Hop has been presented in several publications [9, 8]. We only remind its essential aspects and show some examples that should be sufficient to understand the rest of the paper.

Hop is a Scheme-based multitier functional language. The application server-side and client-side are both implemented within a single Hop program. Client code is distinguished from server code by prefixing it with the syntactic annotation '~'. Server-side values can be injected inside a client-side expression using a second syntactic annotation: the '$' mark. On the server, the client-side code is extracted, compiled on-the-fly into standard JavaScript, and shipped to the client. This enables Hop clients to be executed by unmodified Web browsers.

Except for its new multitier programming style, Hop uses the standard Web programming model. A server-side Hop program builds an HTML tree that creates the GUI and embeds client-side code into scripts, then ships it to the client. AJAX-like service-based programming is made available by service definitions, a service being a server-side function associated with a URL. The `with-hop` Hop form triggers execution of a service. Communication between clients and servers is automatically performed by the Hop runtime system, with no additional user code needed.

The Hop Web application `fib-html` below illustrates multitier programming. It consists of a server-built Web page displaying a three-rows table whose cells enumerate positive integers. When a cell is clicked, the corresponding Fibonacci value is computed on the client and displayed in a popup window. Note the '~' signs used lines *3*,*6*, *7*, and *8* which mark client-side expressions.

```
1: (define-service (fib-html)
2:    (<HTML>
3:       ~(define (fib x) ;; client-side code since prefixed by ~
4:          (if (< x 2) 1 (+ (fib (- x 1)) (fib (- x 2)))))
5:       (<TABLE>
6:          (<TR> (<TD> "fib(1)" :onclick ~(alert (fib 1))))
7:          (<TR> (<TD> "fib(2)" :onclick ~(alert (fib 2))))
8:          (<TR> (<TD> "fib(3)" :onclick ~(alert (fib 3)))))))
```

Let us modify the example to illustrate some Hop niceties. Instead of building the rows by hand, we let Hop compute them. The new Hop program uses the `(iota 3)` expression (line *9*) that evaluates to the list (1, 2, 3) and the `map` functional operator that applies a function to all the elements of a list. The `$i` expression (line *8*) denotes the value of `i` on the server at HTML document elaboration time, seamlessly exported to the client code:

```
1: (define-service (fib-html)
2:    (<HTML>
3:       ~(define (fib x) ...)
4:       (<TABLE>
5:          (map (lambda (i)
6:                  (<TR>
7:                     (<TD> "fib(" i ")"
8:                        :onclick ~(alert (fib $i)))))
9:             (iota 3)))))
```

Before delivery to a client, the server-side document is compiled on the server into regular HTML and JavaScript. It can then be executed by all standard browsers.

3 The HipHop Programming Language

HipHop embeds the reactive primitives of Esterel [1] within Hop while making maximal usage of Hop's expressive power. By convention, the '`&`' suffix is associated with HipHop code. Technically speaking, a HipHop form should be seen in two ways. First, it is a Hop constructor that builds a Hop value that represents a HipHop abstract syntax node. This makes it possible to dynamically build and run HipHop programs from within Hop. Second, it is a temporal statement executed by a *reactive machine* that communicates with Hop using logical HipHop events built by Hop out of physical or programmed events.

The reactive machine is triggered by Hop and perform conceptually instantaneous and deterministic *reactions* to its input HipHop events, generating output HipHop events.

3.1 HipHop Events

HipHop logical events are abstract Hop values of class `HipHopEvent`. They can be inputs and outputs of the reactive machine or local to the HipHop program, then helping synchronization and communication between its concurrent parts. HipHop events have an optional boolean presence/absence *status* and an optional *data value*. The status and value of each event are unique in each reaction and broadcast to the parallel components of the HipHop program.

The status of an event is *absent* by defaut. Input events are set *present* from Hop prior to the reaction using the `hiphop-input!` Hop form; this determines the *input context*. Local and output events are set present from within

the HipHop program by executing the `emit&` statement. The status of an event e is not memorized between successive reactions. It is read using the `(now&` e`)` form, while the status at the previous reaction is read using the `(pre&` e`)` form.

The data value of an event is defined when setting the status, either from Hop using `hiphop-input!` for an input or by `emit&` for an output or local. Contrarily to the status, the value is memorized between reactions. The current value of e is returned by the `(val&` e`)` form, while the value at the previous reaction is read using the `(preval&` e`)` form. As for Esterel, several emissions can occur for the same event in the same reaction; they are said to be *simultaneous*. In that case, the final value of the event is obtained by combining the individually emitted values using a combination function specified in the event Hop object declaration.

3.2 Reactive Machines and their Reactions

Reactive machines interface Hop and HipHop. A machine M is defined by its HipHop input/output logical event interface and its HipHop program.

Hop delivers an input event `A` with value v to a reactive machine M using the form `(hiphop-input!` M `A` v`)`. Any number of inputs can be delivered before a reaction; they are only valid for this reaction. A reaction is triggered from within Hop by `(hiphop-react!` M`)`. Determining when a machine should react is solely Hop's responsibility. However, to simplify a common case, it is possible to write `(hiphop-input-and-react!` M `A` v`)` to pass an input and trigger a reaction right away.

A reaction may trigger output events, the actual output action being performed by associated Hop listeners associated with the events and stored in the reactive machine. To handle data, a reaction may also trigger the evaluation of Hop expressions using the `atom&` HipHop statement, see Section 3.4.

Seen from Hop, a HipHop reaction is simply a standard function call. Seen from HipHop, the execution of the reaction is conceptually performed in zero-delay, the HipHop program sleeping between two successive reactions and remembering its control state from one reaction to the next. This coroutine execution scheme avoids interference between input event registering and reactions, which is a common cause of unwanted non-determinism and deadlocks with classical threading techniques.

A reactive machine can be executed on the server or shipped to and executed on a client, because it is a standard Hop object. Several reactive machines can coexist in the same application, making it possible to use a GALS programming model (Globally Asynchronous, Locally Synchronous) without extra overhead. This will not be detailed here.

3.3 HipHop Intuitive Execution Semantics

The reactive code is based on deterministic sequencing, concurrency, and temporal statements inspired from Esterel [1]. Control positions are memorized from

one reaction to the next. To illustrate sequencing, consider the following sequence:

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

where `A` and `B` (resp. `O`) are input (resp. output) HipHop events. Intuitively, the code waits for `A` and then `B` to be present, before emitting `O` and terminating synchronously: `O` is emitted within the reaction triggered by `B`. Technically, at first reaction, the HipHop control flow stops on `(await& A)`, and yields back control to Hop. HipHop control 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.

To illustrate concurrency, consider now the following HipHop code:

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

Here, `A` and `B` are waited for in parallel, not in sequence. The `par&` statement terminates when all its arms are terminated. Thus, `O` is emitted exactly when the last of `A` and `B` occurs (note that `A` and `B` may be both present in the same reaction if they have been both input into the reactive machine before the reaction is triggered). In HipHop, concurrency and sequencing can be mixed arbitrarily, and the same holds for all other instructions.

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.

3.4 HipHop Core Statements

As for Esterel, statements are divided into *core statements*, which are primitives and handy *derived statements*. Thanks to Hop's reflexivity, derived statements can be trivially defined from core statements using Hop. We first detail the core statements.

The `nothing&` statement does nothing and terminates instantaneously. It is the HipHop no-op. The `(emit&` e $[v]$`)` statement emits its event e with value determined by optional v. It terminates instantaneously. The `(atom&` *expr*`)` statement calls Hop to executes the *expr* Hop expression; it is instantaneous, which means that its Hop argument execution time should be kept negligible in practice.

The `pause&` statement delays execution by one reaction: it pauses for the reaction and terminates at the next reaction.

The `(if&` *test then else*`)` statement instantaneously evaluates *test*. If the result is true, it immediately starts *then* and behaves as it from then on; otherwise, it does the same with *else*. These can be arbitrary HipHop statements. Termination of the `if&` statement is instantaneously triggered by termination of the selected branch. The `seq&` statement executes its arguments in order: the first one 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 sequential repetition of its argument statements, themselves implicitly evaluated in sequence. For instance, `(loop& (pause&) (emit& A))` waits for the next reaction and then keeps emitting `A` at each reaction. Exiting a `loop&` can only be done by using the `trap&`/`exit&`, `abort&`, and `until&` statements, see below.

The `par&` statement starts its arguments concurrently and terminates when the last of them terminates. Therefore, `(par& (await& A) (await& B))` immediately 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 `suspend&` statement immediately starts its body. At all following instants, it suspends (freezes) the execution of its body for the reaction when its condition is true. The `suspend&` statement terminates if its body is executed and terminates. For instance,

```
(suspend& (now& A)
   (loop&
      (emit& B)
      (pause&))
```

emits `B` at first instant and at all subsequent instants where `A` is absent.

The `trap&` statement defines a named exit point for its body. The `exit&` statement provokes immediate termination of the corresponding `trap&` statement, as well as immediate termination of all concurrent statements within the `trap&` body, which do normally receive the control at that instant.

The `local&` statement declares local events in the first argument list. Their scope is the body, which is the implicitly `seq&` list of the remaining HipHop arguments. The declared events are not visible from Hop. A `local&` statement terminates when its body does.

3.5 HipHop Derived Statements

The derived statements can be easily defined from the kernel ones using Hop. The `halt&` statement pauses forever; it is defined as `(loop& (pause&))`. The

`sustain&` statement keeps emitting its event at each reaction. The `await&` statement pauses and waits for its expression to become true and terminates:

```
(define (await& evt)
   (trap& (done)
      (loop&
         (pause&)
         (if& (now& evt) (exit& done)))))
```

The `abort&` statement instantaneously kills its sequential body when its condition becomes true, not passing the control to its body in this reaction; this is what we call *strong abortion*:

```
(define (abort& evt . stmt-list)
   (trap& (done)
      (par&
         (suspend& (now& evt) stmt-list)
         (await& evt (exit& done)))))
```

The `until&` statement instantaneously kills its body when its condition becomes true, but only at the end of the reaction, passing the control to its body for the last time at that reaction as for an exited `trap&`; this is what we call *weak abortion*. The `loop-each&` statement immediately starts its body, and then stongly kills it and restarts it immediately whenever its condition becomes true. The `every&` statement is similar but starts by waiting for the condition instead of immediately starting its body (see [7]).

Once defined, these statements can be freely used in HipHop programs. Note that this makes the language fully user-extensible. One can also build dynamically statements from dynamic values computed during Hop execution, and even dynamically modify the program between two reactions, for instance to use and orchestrate services dynamically detected at runtime. Note also that there is no need to redefine the basic arithmetic, list, and string expressions since Hop's ones can be reused (with some care however, no details given here).

4 An Application Example

We build a Lastfm-like smart music player called HopFM that orchestrates musical content and related information available on third party Web sites. It plays music continuously, switching from one artist to another according to musical similarities. It automatically fetches and displays information about music and authors.

The user first selects a musical genre in a dynamically discovered list. This activates the Hop control screen of Figure 1. The top of screen is used to adjust the volume, pause the music, switch to the next track, etc. Below stands the Start button (zone 2, Figure 1) that starts HopFM when clicked. HopFM then searches the internet for a random artist of the selected genre, downloads a limited number of tracks of this artist, and starts playing. When a new track starts,

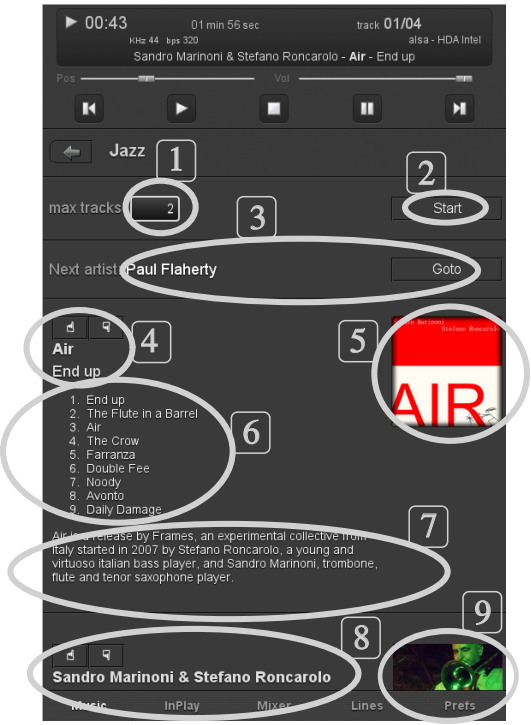

Fig. 1. The smartphone screenshot of HopFM. Ellipses show active zones managed by Hiphop. Zone 1 sets the number of tracks to play per artist. Zone 2 initiates the music play by starting the reactive machine. Zone 3 displays the name of the next artist and the button to switch to his music. Zone 4 reports the current track name and the album name; the two top buttons enable the user to emit positive or negative recommendations. Zone 5 displays the album image, zone 6 the album's track list, zone 7 comments about the album. Zone 8 displays the artist name and two buttons to emit positive or negative recommendations. Zone 9 displays an image of the artist.

HopFM looks for related information on the internet: the associated album, its image, its track list, and reviews. In parallel, HopFM searches and displays information about the played artist: his biography, his discography, some news, and some blogs and reviews.

Also in parallel, the application searches the Web for similar artists to be played later on, either when the currently downloaded tracks are over or when the user clicks the Goto button (zone 3). If no similar artist is found, or if all the known similar artists have been played, HopFM randomly chooses a new random artist of the genre and starts again. HopFM keeps running until the user clicks the top stop button.

The music player uses the following third-party services to retrieve musical contents and information about the tracks and artists: *Free Music Archive*[3] (FMA), which provides mp3 music and informations about the tracks, albums, and artists; *EchoNest*[4], a database about tracks, artists, images, artist similarities, and alignment information between other databases; *MusicBrainz*[5], an open music encyclopedia that collects music metadata.

HipHop orchestrates the requests to these remote services. For instance, to search for an artist image, several requests to several sites are emitted simultaneously. As soon as the first one completes, the other ones are aborted and the GUI is updated. If the artist changes before any request completes, all current requests are aborted.

HipHop also handles user interactions and signals raised by the actual hardware music player. Web services returns or errors, user interaction, and music player events are the three sources of external asynchrony. They are all managed uniformly by HipHop, local synchronous parallelism and communication being essential to regulate and orchestrate external asynchrony.

4.1 HopFM Implementation

Clicking Start invokes the Hop client-side `hopfm-play` function, which constructs a reactive HipHop machine, creates various events, binds the external Hop events to the HipHop interface events, creates a HipHop reactive program, loads it into the machine, and eventually triggers the first reaction:

```
(define (hopfm-play catalog genre::genre)

   (define musicstate (instantiate::HipHopEvent))
   (define track (instantiate::HipHopEvent))
   (define artist (instantiate::HipHopEvent))
   (define playlist (instantiate::HipHopEvent))
   ...

   (define (hopfm&) ...)

   (let ((M (instantiate::HopHifiMachine
               (program (hopfm&)))))
      ;; bind the machine to external HopHifi events
      (add-event-listeners! M musicstate track artist)
      ;; trigger first HipHop reaction
      (hiphop-react! M)))
```

The Hop function `add-event-listeners!` connects the actual hardware player to the HipHop program. It binds Hop listeners that forward the events to HipHop.

[3] `http://freemusicarchive.org`.
[4] `http://echonest.com`.
[5] `http://musicbrainz.org`.

```
(define (add-event-listeners! M musicstate track artist)
   (add-event-listener! server "hophifi-state"
      ;; listener called when the music player state changes
      (lambda (e)
         ;; forward the Hop event to HipHop
         (hiphop-input-and-react! M musicstate (event-value e))))
   (add-event-listener! server "hophifi-track"
      ;; listener is called when a new track starts
      ;; or when the playlist changes
      (lambda (e)
         (let* ((ev (event-value e))
                (tk (list-ref ev.playlist ev.song)))
            ;; forward the track and artist to HipHop
            (hiphop-input! M track tk)
            (hiphop-input! M artist (track-artist tk))
            (hiphop-react! M)))))
```

The HipHop program `hopfm&` runs a number of components in synchronous deterministic parallel, each in charge of a specific task. These components synchronize each other by communicating synchronously using the HipHop events defined above: `track`, `artist`, `playlist`, `album`, etc. The `random-playlist&` component looks for random playlists, each playlist being associated with an artist; `playlist&` waits for playlist changes and starts searching for the next artist; `track&` waits for the hardware player to start a new track, checks if it belongs to a different album or to a different artist and, in this case, emits the HipHop events `album` and `artist` towards the other components; `artist-info&` waits for a new artist event, searches the internet for information about that artist, and emits an event that lets the `gui&` component update the screen. The HipHop program stops when the user clicks the main stop button, which raises the `musicstate` HipHop event with value `stop`; the enclosing `until&` statement then generates a global preemption that kills all internal activities and terminates; termination can also occur if no artist is found (`musicstate` value `ended`).

```
(define (hopfm&)
   (until& (memq (val& musicstate) '(stop ended))
      (par& ;; running all the components in synchronous parallel
         ;; peek a random playlist
         (random-playlist& catalog genre playlist)
         ;; playlist manager
         (playlist& playlist)
         ;; deal with new tracks
         (track& track album artist)
         ;; manage new artists
         (artist-info& catalog genre artist bio discog similar playlist)
         ;; update the gui
         (gui& musicstate track album artist bio discog similar))))
```

Let us detail `random-playlist&`. Its operation requires two steps: calling FMA for a random artist of the desired genre and checking that this artist has published music. The FMA request is proxied via the Hop server using the HipHop

`with-hop&` statement that takes as parameter a service call and a HipHop event to emit if the request completes successfully; `with-hop&` simply terminates silently otherwise. Note that the artist found is kept local, since the global artist handled by other modules is the one currently played.

```
(define (random-playlist& catalog genre playlist)
   (trap& found
      ;; start looping
      (loop&
         ;; creates two local events
         (local& ((local-artist (instantiate::HipHopEvent))
                  (local-playlist (instantiate::HipHopEvent)))
            ;; get a random artist from FMA
            (with-hop& ($hopfm/genre/artist/random genre catalog)
               local-artist)
            ;; get the tracks of that artist
            (with-hop& ($hopfm/artist/tracks (val& local-artist))
               local-playlist)
            (if& (pair? (val& local-playlist))
               ;; an artist with music is found
               (seq&
                  (emit& playlist (val& playlist))
                  (exit& found)))))))
```

The `artist-info&` component searches in parallel an image of the current artist, information about that artist, and a similar artist with a playlist. it outputs `bio`, `discog`, `similar`, and `playlist` towards the other components as soon as the corresponding information has been found.

```
(define (artist-info& catalog genre artist bio discog similar playlist)
   (every& (now& artist) ;; we have a playlist for that artist
      (par&
         ;; request a similar artist list
         (similar-artist& catalog genre playlist artist similar)
         ;; fetch artist biography and discography
         (artist/bio& artist bio discog)
         ;; fetch the artist images
         (artist/image& artist))))
```

The `artist/image&` subcomponent of `artist&` calls FMA and EchoNest in paralle to find an artist image. As soon as one server responds, the other request is aborted using a `trap&` statement with `exit&` triggered when the `img` local signal is received. If no image is found, the currently displayed image is hidden:

```
(define (artist/image& artist)
   ;; find the first image out of two services
   (let ((el (dom-get-element-by-id "hophifi-internet-artist-image")))
      (local& ((img (instantiate::HipHopEvent (name "image"))))
         ;; try to find one image on two different servers
         ;; abort the pending request as soon as one returns
         (trap& (done)
            (par&
               (seq&
                  (with-hop& ($hopfm/artist/image (val& artist)) img)
                  (if& (now& img) (exit& done)))
               (seq&
                  (with-hop& ($hopfm/artist/image/echonest (val& artist))
                     img)
                  (if& (now& img) (exit& done)))))
         (if& (now& img)
            ;; update the GUI with the new image
            (atom&
               (node-style-set! el :visibility "visible")
               (set! el.src (val& img)))
            ;; no image was found, hides the current one
            (atom& (node-style-set! el :visibility "hidden"))))))
```

Note the architectural power of nested preemption. In `artist&` above, when the `artist` signal event is received, the `every&` preemption loop kills is body, aborting `artist/image&` and by transitivity its spawned `with-hop&` requests that might still be pending. Furthermore, the enclosing `until& musicstate` statement of `hopfm&` has an even greater preemption power since it kill all activities in the program: external preemptions dominate internal ones.

The other HopFM HipHop components are omitted here because their implementation is similar.

5 Related Work

We presented a preliminary version of HipHop at the Plastic'11 workshop [2]. While the core language has been kept stable, the integration with HOP has been entirely redesigned and the former version should be considered as obsolete.

Orc [3] addresses the service coordination issue by proposing a combinator-based process calculus The temporal algebra of HipHop is richer than that of Orc. However HipHop does not yet offer the flexibility of the Orc data-stream pipeline $f > x > g$ operator for large-scale data processing. Flapjax [5] provided a unified framework for client-side event-based programming, based on implicit control defined by data streams instead of explicit control in Hiphop. Jolie [6] is a framework to write and orchestrate Web Services using a service-oriented programming language inspired by the π-calculus. However, contrarily to HipHop, Jolie is limited to server-only orchestration.

6 Conclusion

We have presented HipHop, a new domain-specific synchronous language geared to the orchestration of services and user intreaction within Hop on server and client sides. HipHop deals with logical events exposed by Hop. Its statements are imported from Esterel. They are based on temporal sequentiality, concurrency and preemption, which make it possible to replace the traditional asynchronous thread / event-handler spaghetti [4] by a well-understood synchronous programming style imported from embedded systems programming. The reflexivity of Hop makes it possible to build HipHop programs, ship them to clients or other servers, and run them.

With our LastFM example, we have sketched how to orchestrate Web services and GUI events with HipHop, using its reactive statements as key architectural tools.

Our current implementation is an interpreter directly based on Esterel's constructive semantics [1]. More efficient implementations will certainly be needed for large-scale applications, see [7].

Acknowledgements: we thank Cyprien Nicolas, who implemented HipHop in Hop and participated in the PLASTIC'11 first paper about HipHop.

References

1. G. Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
2. G. Berry, C. Nicolas, and M. Serrano. HipHop: A Synchronous Reactive Extension for Hop. In *Proceedings of the PLASTIC'11 workshop*, Portland, USA, Oct. 2011.
3. 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.
4. E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.
5. 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.
6. F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. Jolie: a java orchestration language interpreter engine. *Electr. Notes Theor. Comput. Sci.*, 181:19–33, 2007.
7. D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.
8. M. Serrano and G. Berry. Multitier Programming in Hop - a first step toward programming 21st-century applications. *Communications of the ACM*, 55(8):53–59, Aug. 2012.
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.