Synchronous-Reactive Web Programming

Rémy El Sibaïe
Sorbonne Universités,
Univ Paris 06, UPMC, CNRS,
LIP6 UMR 7606,
4 place Jussieu 75005 Paris.
remy.el-sibaie@lip6.fr

Emmanuel Chailloux
Sorbonne Universités,
Univ Paris 06, UPMC, CNRS,
LIP6 UMR 7606,
4 place Jussieu 75005 Paris.
emmanuel.chailloux@lip6.fr

Abstract

The current event-based model of web client programming lacks of a high level abstraction for concurrency and communication when many interactions are involved. The design of JavaScript runtime in the browser is very simple and chains steps of input handling and computation of output in a sequential way. This definition describes exactly a subset of programs well handled by the synchronous-reactive model. It proposes constructs to express parallel tasks communicating through broadcasted signals enforcing a static hypothesis of determinism, coherency and causality that improve programs composition. It is then interesting to consider client events as inputs and web view elements as outputs of a synchronous-reactive program. We describe here the design of `pendulum`, a language extension implementing those principles and targeting web client programming, which generates fast sequential code.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Interoperability; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors

General Terms Design, Languages, Experimentation

Keywords Web client programming, Synchronous-reactive programming, Functional programming, OCaml, Embedded domain specific language, Static typing

1. Introduction

The web programming community is currently shifting the state of client programming to more evolved languages. It includes new features and libraries in JavaScript, and new innovative languages using JavaScript as a backend. The goal is to improve both safety and expressiveness. The former is usually achieved by static analysis like Typescript [1] and Flow [13] and the latter by adding new constructs to better compose and orchestrate the interactions as in React [14] or Elm [9], both data-flow models. This is possible and supported by the fact that the JavaScript runtime model is simple and sequential, even for interactions handling.

REBLS'16, November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4644-3/16/11...$15.00
http://dx.doi.org/10.1145/3001929.3001931

1.1 Weaknesses of Interaction Handling in Common Event-Based Systems

A client program is continuously computing outputs (communications, displays) from the inputs (hardware, communications). It has to handle a tons of interactions seen as events and their reaction.

The JavaScript model is driven by callback functions attached to these events. We usually want to express complex things like waiting for F5 key, or CTRL-R button to be pressed to refresh the window, except if a video is currently playing. This implies a logical computation of the events like $(F5-Key \vee (CTRL-Key \wedge R-Key)) \wedge not\ PlayingVideo$, which is hard to write in an event-based system as simple as in JavaScript without using an explicit global environment shared by callback functions, which is painful to maintain and compose with new features.

Functional reactive programming (FRP) [11] and other recent data-flow models like React fit well to solve this kind of interaction as it uses a global environment implicitly built and modified by the composition of continuous values usually called signals or behaviors. Describing the expected result builds both the data dependencies and the calculus which leads to less error in general.

In some cases, FRP is not the best way to express an interaction. For instance, modifications on a text field is easy to map in FRP because it is a continuous information. On the opposite, for some events, we want to express control more than data computation and it is less natural to express it in FRP.

1.2 Synchronous-Reactive Programming

The synchronous-reactive model [15] (SR) comes from three main languages: Esterel, Signal and Lustre and their derivatives. In this paper, we only focus on Esterel [4], which is an imperative reactive-synchronous programming, when the two others are dataflow oriented.

As in reactive programming, the execution of an SR-program is divided among steps. There is a single engine running the whole program called the *clock*, which decides the frequency of the steps. The tasks in parallel are thus said to be synchronized and communicate by broadcasting *signals*, that are either present or absent. The language components are statements modifying a global environment by side effects. Those statements can be instantaneous (execution in one step) or not (execution in more than one step). The program is compiled in such a way that the execution is fully sequential. It takes the inputs before a step, computes them, and generates the outputs at the end.

The SR model is also based on the *synchronous hypothesis*. It states that, since every communications and computations are instantaneous, it is possible to detect the incoherences or any causality error. This makes the program behavior safe and free of data-race or interblocking. This checking is achieved by causality analysis of the program. The safety provided by the SR model is not free as there are static constraints on programs. First, it is not possible to execute a potentially infinite recursion in the language itself and the only way to repeat is to gather information from one step to the next. Second, the causality analysis can fail and rejects a program that seems correct to the programmer. Those two constraints are balanced by the abstractions brought by the language constructs. In the end, if a particular code cannot be written in the SR-language, it is still possible to write it in the host language (like `C` for Esterel) at the cost of losing causality analysis of this part.

1.3 Contributions

The sequential computing of a set of outputs from a set of inputs is exactly what a web application does. That is why we, in this paper, state that synchronous-reactive model is fully applicable to the web client. It brings additional constructions for concurrency with the parallel operator and the imperative approach helps to represent JavaScript events as signals. The automatic static scheduling of parallel tasks saves a lot of work for the programmer so s/he can focus on something else and it makes the program easier to maintain and compose afterwards.

To support those statements, we introduce the SR language `pendulum`, which is an immersion of a subset of Esterel into a Web client context. We also extends this model to smoothly connect the clock of the SR programs to the events of the Web and their reaction. Then we show how to use the clock to take advantage of the Web client runtime. The implementation is a syntactic extension over OCaml and the OCaml code can be then compiled to JavaScript.

We start by introducing the core constructs of the language and the programming API between the host language and `pendulum` in **Section 2**. The new contributions to mix Web and SR programming with the results are detailed in **Section 3**. **Section 4** presents the compilation scheme use to generate OCaml code from SR programs. After a longer example in **Section 5**, we compare our approach to related work in **Section 6** and we conclude.

2. Overview of `pendulum`

We introduce the language `pendulum` as a subset of Esterel semantically speaking. It brings the complete SR expressiveness: communications, repetition through instants, parallel and escaping. A `pendulum` program is then a combination of statements with input or output signals parameters. The syntax is given in Figure 1. We separately refer to **`synchronous`** keywords in darkblue (dark grey) and **`OCaml`** keywords in darkviolet (light grey) on the first occurrences of each keywords in the text.

2.1 Basic Features

Communications are done through **`emit`**. It broadcasts the presence information of a signal with its new inner value in its scope. A global signal, defined by **`input`** or **`output`**, reaches the whole program but a local signal, defined by a **`let-in`**, is visible only in a particular statement. The signal stays present for the current step and switches to absent at the end until the next emission. If the signal is carrying a value, it is kept between steps. The statement **`present`** choose one of its branches depending the result of the presence test.

prog ::= *header** *stmt*
header ::=
| **input** *ident*;
| **output** *ident*;
test ::= *ident*
stmt ::=
| **emit** *ident ocaml-expr*
| **nothing**
| **pause**
| **present** *test stmt stmt*
| **loop** *stmt*
| *stmt* **||** *stmt*
| *stmt* **;** *stmt*
| **let** *ident* **=** *ocaml-expr* **in** *stmt*
| **trap** *label stmt*
| **exit** *label*
| **!** *ocaml-expr*
| **suspend** *test stmt*

Figure 1: `pendulum`'s syntax

To execute several statements in sequence, we use the operator `;` as in imperative languages. When the left operand terminates, the right operand is started in the same step. The repetition is handled by the **`loop`** statement, executing indefinitely its body but not more than once per step, as explained in Section 1.2. To express where the step stops and waits for the next, the programmer has to write **`pause`** himself. It is the case because a `loop` statement could have a pause at the beginning, end, middle or even spend several steps in the body of the loop. If `pause` is missing, it can be inserted by the compiler at the end of the body so that the program does not actually loop instantaneously at runtime. `pause` is also used outside of this context to wait for the next step to continue. To finish , **`nothing`** does nothing, and the `!` operator runs a host language expression instantaneously.

The reader may have noticed there is no mention of types in the syntax. The task of type checking and type inference is left to the host language as explained in section 2.3.

2.2 Parallel and escaping behaviors

The purpose of the `||` operator is to evaluate left operand and right operand in parallel. Their execution are synchronized on the same clock and their steps are executed at the same time. The execution order is specified by the compiler during the scheduling pass and is constrained by the communications between those, thus the execution of the parallel is deterministic.

In `pendulum`, it is also possible to escape the control-flow with a **`trap`** statement. When an **`exit`** statement with the same label is executed in its body, it terminates immediately and the control-flow continues the execution after the `trap`. If escaping blocks are nested and several `exit` are executed, it terminates the body of the outermost `trap`. If there are parallel tasks in a `trap` and one calls `exit`, all of them are terminated. The last statement, **`suspend`**, blocks its body each step a signal is present.

2.3 Implementation

The implementation is built on top of OCaml[16], a general purpose language, embedding the functional, the imperative and the object programming paradigms. The embedded code is compiled thanks to the PPX macro engine, part of the tools distributed with OCaml. We choose this language for its several powerful features: algebraic datatypes, pattern-matching, exceptions, modules, objects..., but also for its typechecking and type inference that bring both safety and expressiveness. It is well-fitted for the Web too, as the compiler `js_of_ocaml` [23] generates vanilla JavaScript code from OCaml bytecode. In this paper, JavaScript and OCaml are considered to be interchangeable as one compiles to another and the Document Object Model (DOM) library is fully accessible in OCaml. They are just referred as *the host language*. We give an overview in Figure 2 of the compilation process to situate our work in the compilation chain.

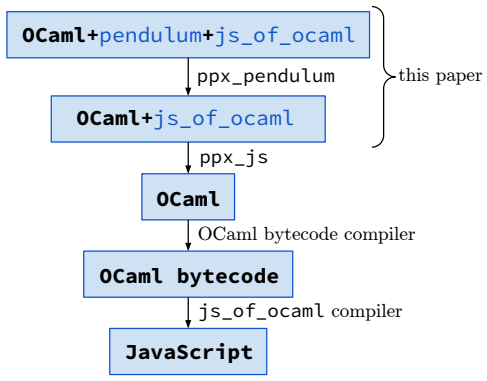


Figure 2: Global compilation process

2.4 Interface with host language

A synchronous-reactive program runtime engine is strongly different from a common imperative program. Indeed, the execution of the successive steps has to be started from the host language runtime. This is called the clock, which decides at what rhythm the program is running. Before exploring more possibilities, the model is kept simple and low-level by exposing a programming interface to simulate the clock.

In the following example, `p` is defined as a synchronous program by the special syntax in OCaml, `let%sync`, which is an entry point for the macro engine. `p` awaits for `s1` and `s2` to be present at the very same step. Then it emits a signal `o`. The new value of `o` is computed in the OCaml world (see Figure 1) by string concatenation (`^`) of the values of `s1` and `s2`. The `!!` operator serves this purpose by extracting the OCaml value of a `pendulum` signal in an OCaml expression. The inferred type is thus `string` for all the three signals.

```
let%sync p =
  input s1, s2;
  output o;
  loop (
    present s1 (present s2
        (emit o (!!s1 ^ !!s2)));
    pause)
```

The program `p` is an object in the OCaml program with a method `create`[1] to instantiate a new version of it, p0, and a new environment. This method takes as much arguments as the number of input and outputs signals. Input signals takes the initial values and the outputs signals a pair of initial value and callback. Here, the function `print_endline` is called whenever `o` is present.

```
let p0 = p#create ("", "") ("",
    print_string)
```

The object `p0` has a method `react` that starts one step of execution. It also has a method for each input signal that sets

[1] # stands for methods call for objects in OCaml

the signal value and switches its status to present for the next step.

```
p0#s1 "hello ";
p0#s2 "world !";
p0#react
>>  "hello world !"
```

Now both input signals are set as present with new values and one executes a step of the program (the reaction), computing the outputs. The callback assigned to `o` is also called at this point. When `react` ends, the signals are switched back to absent.

3. Mixing Synchronous with Asynchronous Event-Base Programming

A synchronous-reactive program clearly states the dependencies between the inputs and outputs of the system. It is a closed box where the inner communications are checked.

At this point of the paper, `pendulum` is still not related to web programming and can be used in a general purpose OCaml program, but the programming layer introduced in section 2.4 can still be error prone as the user has to deal with triggering each step of the computation directly in the program.

3.1 Input and React in Callbacks

Input is the action of setting a signal present for the next step and react is the action of triggering the step. Thus the naive way to connect an SR program to an event system is to use a signal for each event and to *input* and *react* in the callback of the event.

```
let sp = createSpan document;;
appendChild body sp;;
let%sync mouse_loc =
  input location;
  loop (
    present location !(
      sp##.textContent :=
        Js.some (Js.string location)
    ); pause
  );;
let m = mouse_loc#create ("");;
(* Input and react *)
window##.onmousemove := handler (fun ev ->
  m#location (sprintf "%f, %f"
    ev##.clientX ev##.clientY);
  m#react
);;
```

Figure 3: Program displaying the mouse location

The code in Figure 3 presents this methods using the `js_of_ocaml` library[2]. It prints the location of the mouse in an HTML span tag. It executes the body of the loop at each instant. If the signal `location` is present, it prints its content on a span by side effect.

The goal in the following section is to add constructs that help to generate the *input-and-react* part of the code behind the scene, reducing boilerplate and smoothly connect the SR world and the Web.

3.2 Events as Inputs, DOM Element Properties as Outputs

The inputs of a web client program are clearly defined by the DOM API as events. Those events are generally spawned from a target that can be for instance, the `window` (the browser), the `document` (the page), an `XmlHttpRequest` or any DOM element. Those inputs can then be expressed as a pair $(event, element)$. Our proposition is to introduce in the SR language the DOM elements as parameters of the program and the possibility to use their events as input signals, and their properties as output signals. We extend the previously defined syntax of `pendulum` *header*, *test* and *stmt* in the way shown Figure 4.

header ::= **element** *ident gather*? **;**
test ::= *ident* | *ident***##***event*
stmt ::= ... | **emit** *ident***##.***property ocaml-expr*
gather ::= **{** (*event* **=** *ocaml-expr***,** *ocaml-expr***;**)* **}**

Figure 4: Extended syntax

It is now possible to refer to an event that could spawn on an element by mapping a signal on it. We can write it in the following way, where `window` has been defined as an element parameter of the program:

```
present window##onmousemove ...
```

This signal, `window##onmousemove`, is a now a read-only global input of the program. This code generates a callback that inputs the value of the signal for the next step and call the `react` method each time the event is spawned. At this point, the clock of this program has the spawn frequency of the event. If several events are referred, they all trigger a step and the frequency of the clock become the frequency of the reunion of those events.

By default, the signal `window##onmousemove` carries a value of type `event option`[3] to represent the state where the event has still not spawn, where `event` is the common JavaScript event object. This is a bit trickier to interact with option types, so we add a construct to initialize and gather (Figure 4, rule *gather*) event values when defined. This feature is also very useful for the programmer to give details on

[2] `##` (darkviolet/lightgrey) stands for JavaScript method call, and `##.` for JavaScript property access and modification. The object model of JavaScript is not treated in the same way than the one of OCaml in this library. Do not confuse with `##` (darkblue/darkgrey) in `pendulum`.

[3] Types must be read from right to left in OCaml. Here we have a signal containing an option containing potentially an event

how the event must be input to the SR program, and even more importantly, it is a first step to handle several events between two steps and compose them.

Our inputs can be clearly specified, but it is also interesting to investigate what can be done with outputs, as we want a cleaner way to directly modify output data without unchecked side effects (using `!`). With the possibility to `emit` directly into JavaScript properties (Figure 4, rule *stmt*), manipulating the DOM becomes part of the SR program and can be checked in the same way signals are.

In Figure 5, we use all the previously defined extensions to rewrite the program in Figure 3. The call to `react` is removed and becomes implicit, so is the call to input the signal. The value of `window##onmousemove` signal is defined by a string format of the float coordinates of the mouse.

```
let%sync mouse_loc2 =
  element sp;
  element window {
    onmousemove = "", (fun ev ->
        Js.some (Js.string (
          sprintf "%f, %f"
          ev##.clientX ev##.clientY))
      );}
  loop (
    present window##onmousemove (
      emit sp##.textContent
        !!(window##onmousemove)
    ); pause)
 let m2 = mouse_loc2#create (sp, window)
```

Figure 5: Program displaying mouse location with the extended syntax

In the next section we will explain how this model and those extensions generate pretty fast programs without being intrusive in the SR code.

3.3 Browser animation frame as a clock

The web browser is a very fast rendering machine whose job is to refresh the display as soon as updates are made in the DOM. Modifying the DOM in every callback can lead to poor performance since it forces the browser to redraw the display several times. An addition has been made to the client API several years ago to work around this problem: `requestAnimationFrame` [24]. This method takes a function as a parameter that will be called just before the next redraw. By using a recursive call, it is also possible to loop at the rhythm of the browser. *Batching* the updates to the DOM and applying them just before the next redraw allows the browser to optimize its execution and reduces the number of redraw. This technique is used by Elm [8] and Mercury [22].

It is actually possible to make the SR runtime take advantage of this execution scheme. Instead of both setting the input signal and react in the callback, it is possible to input only and deal with the reaction in the next `requestAnimationFrame`. Doing this is not intrusive in the code, meaning there is no need to modify the entire program to change the clock from an event-based clock or on a requestAnimationFrame-based clock. Here, it is just a toggle option to the SR program (`~animate`). If there is a lot of events that input the same signal, the value is just gathered without calling `react`. The reactions, and thus DOM updates, are only called just before redrawing the DOM elements in only one big update. If the step takes too long and misses the next redraw, the result will just be postponed to the one after.

We measured the time performance of this execution scheme. We used the well-known comparison benchmark TodoMVC [21], which describes an implementation for a todo-list application. Designed to compare client programming frameworks in expressiveness, maintainability and lines of code, there is also an already existing speed benchmark on TodoMVC applications from Mercury. This benchmark executes *add 100 item*, *select each* and *remove each*. Those actions corresponds to one hundred events generated in a row and the tested program has to react to them. We compare the average execution time in milliseconds to display the result on 30 runs for `pendulum` and several other web frameworks (Figure 6). These tests are executed on Google Chrome 51 with an Intel Core i7-4690

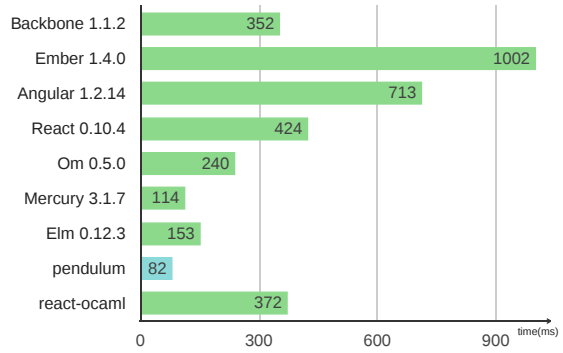


Figure 6: TodoMVC speed benchmark over 30 runs (time in ms)

We can see that Mercury, Elm and `pendulum` have rather good results compared to other frameworks that do not batch DOM updates. In `pendulum`, all updates are first gathered in a list signal. After, during the animation frame following the events, all updates are applied in one hit. Beware that this benchmark does not demonstrate that those frameworks are better in general, but mostly that they can handle efficiently a lot of event occurrences spawned at the very same *time*. It happens but it is not the common context for a web application. However the notable conclusion for `pendulum` is that the compilation from SR to sequential JavaScript code does not add any overhead in execution time, in particular for the

`react` method, and it can have a reasonable speed without changing the code.

4. Compiling Through Control Flow Graphs

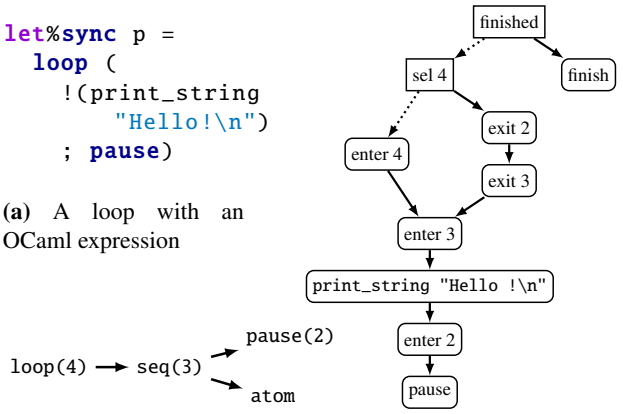
There are several ways to compile and execute Esterel code using state automata which explodes in generated code size, using circuit semantic slower than using automata, and using control flow graphs (CFG). We choose the last option, which offers a good compromise between efficiency and code size. This compilation technique is detailed in the Compiling Esterel book [19]. Another resource with many details on compiling `pendulum` is the previous paper of the authors [10] (it is in French but compilation rules are not).

GRaphCode (GRC) compilation constructs a control flow graph of the steps of the program. The choice of the executed step depends on two environments, the selection tree and the input signal set. The selection tree references all the statements with their current state.

There is only two states for a statement: either not started (finished) or in pause waiting to be resumed. Thus, the compilation takes an SR statement p as input and builds two CFGs, the *surface* being the CFG of the first execution and the *depth*, being the CFG of the resumed execution of p. CFGs of every sub-statements are then composed recursively in a bigger one that is the step function. Control flow graph primitives modify the selection tree and the signal environment during the execution by *entering* or *exiting* statements, and also testing the current state of a statement to know if it is the next to execute or not.

```
let%sync p =
  loop (
    !(print_string
        "Hello!\n")
    ; pause)
```

(a) A loop with an OCaml expression



(b) Selection tree of p **(c)** Control flow graph of p

Figure 7: Control flow graph transformation example

For instance, the program in Figure 7a has its selection tree in Figure 7b (same structure as the syntax tree). There is an infinite `loop`, so, at each step, the only final possible node is pause. A header is added at the top of the graph to keep the information of the end of the program.

This technique is the best choice in our case as there is a whole-program, static causality analysis before this phase. We can then benefit from this static phase to generate efficient, sequential OCaml code from the control flow graph, resulting in the method `react` previously introduced.

5. Example: Reactive Media Player

To better understand the SR model and the additions made in `pendulum`, we will show in this section an example of a reactive media player in a few lines of code, where the behaviors of the controls are written in `pendulum`. The goal is to give the intuition of how to solve a common interaction problem with it. The full example can be found on `pendulum`'s page[4].

We will start by a simple HTML code, containing an audio tag (`media`), a button (`playbtn`) to switch between play and pause, and an input tag (`progress`) typed as `range` to use as a progress bar and seeker, as shown in Figure 8.

```
<audio id="media">
  <source src="aerosmith.mp3" type="audio/mp3">
</audio>
<button id="playbtn">Play</button>
<input type="range" id="progress" value="0"/>
```

Figure 8: Screen capture of the player

We start describing the SR program orchestrating our audio player. It is called `reactive_player` with three element parameters corresponding to each HTML tags described previously. There is also a local signal containing the current boolean state of the video (`true` for playing). Each time the user clicks on the button the value of the state is switched and `update_state` is called. This function modifies the DOM by calling the method `play` or `pause` on `media` and switch the display on the button as expected.

```
let%sync reactive_player =

  element playbtn;
  element media;
  element progress;

  let state = media##.autoplay in
  loop (
    present playbtn##onclick

      (emit state (not !!state));
    present state !(
      update_btn_content playbtn !!state
    ); pause
  )
  || (* ... *)
```

[4] https://github.com/remyzorg/pendulum

Now, we want to handle media progress updates in parallel of the previous part. The DOM API specifies that the event `timeupdate` can be used to react to the timer of the video. This is the one we are using as a signal.

Figure 9: Glitch: the cursor is jumping while seeking

However the progress bar must not be updated if the user is seeking, or the result will be like in Figure 9. So, we add a local signal `seeking` with this information and we do nothing if it is present. If it is absent, we update the value of the progress bar depending on the media current time.

```
let seeking = () in
loop (
  present media##ontimeupdate (
    present seeking nothing
      !(update_slider progress media)
  ); pause
)
|| (* ... *)
```

Now, we must implement the other part and which consist of updating the time of the media depending on the progress bar chosen value after seeking, once again in parallel. We start by waiting for the user to press the mouse on `progress`. Then we define an escape block with label `release`. After, at every step, the program emits `seeking` until the mouse button is released (`mouseup`).

```
loop (
  await progress##onmousedown;
  trap release (
    loop (
      emit seeking;
      present progress##onmouseup (
        !(update_media media progress);
        exit release
      ); pause
    )
  ); pause)
```

At this point the program updates the current time of the media and calls `exit release` to terminate the `trap release` block, the body of the loop as well, and restarts by waiting for `mousedown`. Then we just have to instantiate and start this program from OCaml.

In this example we can clearly see the advantages of the `||` operator to compose different parts communicating without the need to think about the execution order. It allows the programmer to build the code in an incremental way, and when it is compiled we know that the causality is correct. The proposed extensions also simplify the merge between two worlds and help the programmer to stay at a higher level of abstraction.

6. Related work

6.1 Synchronous-Reactive Languages

The domain of synchronous-reactive programming has seen several new ideas since Esterel and some are close to `pendulum`. Our ideas are inspired by ReactiveML [17] and SugarCubes [6]. The former is a dedicated language for SR programming. It is strongly typed and targets OCaml for compilation. This model, as SugarCubes, differs from Esterel because it is not possible to react to the absence of a signal. This constraint makes programs correct by construction with a different (and not weaker) expressiveness. ReactiveML is also different at the runtime and compilation level. The scheduling is dynamic and the code is generated from an intermediate language of continuations. ReactiveML is not especially dedicated to the Web, and was not designed to handle our modifications to the runtime.

In the SR community, there is an other initiative to execute Esterel-like program in a client-server web application, on which this work partially relies. It is called HipHop [5] and it is based on the multitier language Hop [20]. Hop and HipHop have the legacy of Scheme programming languages and tend to stick to the Web in terms of dynamic typing, and reflexivity. HipHop doesn't bind JavaScript events to signals thus the clock must be handled manually when `pendulum` has the element feature introduced in section 3. Another difference comes from the runtime model: reactive programs are not compiled in HipHop but built as JavaScript objects and interpreted. It even allows the programmer to dynamically add statements, which is powerful but could possibly leads to runtime errors, what we want to avoid in `pendulum`. The execution model also implements the circuit semantics of Esterel, which generates shorter but less efficient programs than GRC does.

6.2 Discussion about FRP and Dataflow

To pursue the introduction, web client programming is in constant mutation. New languages and frameworks spawn every month with incredible new ideas. Data-flow models are currently a success, by taking their roots in long term research in functional programming languages.

Elm [9], a good example, is a reactive language where the web client document is described as a combination of continuous values. The composition of those signals cannot create a cyclic definition as everything is functional and immutable with static typing. Elm has also a rich API that redefines all interactions in terms of signals. It makes web programming both elegant, safe and fast. There are other dataflow libraries in JavaScript, like Flapjax [18], implementing the functional reactive paradigm, and even React [14] is a form of functional reactive programming. OCaml has also its FRP library called React (another one) [7], which is heavily used by the Eliom Framework [2] to build multitier applications.

As stated before, for some cases, FRP does not have a natural expressiveness for programs where the problem re-

lies more on a state automata than on a continuous computation of a result. Imperative SR programming and FRP seem to solve a common subset of problems, but in fact, they are both efficient in different part of the program and could be combined to extend the expressive power of a global overhaul model, that can express both control and data. This is what SCADE [12] does, the industrial heir of Lustre and Esterel, by using both state automata and continuous signals.

7. Conclusion

In this article we introduced `pendulum`, a language extension for synchronous-reactive programming. We showed that this model properly addresses the problem of concurrency and interaction handling in web clients. Above that, new constructs added to the original Esterel make possible to unify the different worlds in a smooth way where communications are directly mapped to JavaScript events. The generated code is fast to handle and reacts to events, by using ideas, coming from web reactive programming to update the DOM, that can be directly applied to SR model.

Future work might involve a rework of static scheduling that is currently too simple and rejects valid programs during static analysis. More work is also necessary to compose SR programs in a host language in presence of static scheduling. An other limit of the language is the `element` construct that is currently constrained to static element but that could be extended to a collection of the same kind of element, so we would be able to represent a dynamic list of it. An experiment is scheduled in the near future to use `pendulum` in the context of a multitier framework like Eliom, in combination with FRP. This leads to explore a way to execute SR programs both on the client and the server, where communications are deeply asynchronous. To finish, an investigation on how `pendulum` could interact with common JavaScript frameworks [3] seems necessary to communicate with web programmers.

Acknowledgments

Many thanks to the reviewers for their attentive reading and their numerous remarks. Thanks to Vincent Botbol and Pierre Talbot from UPMC for the helpful proof reading and their advices.

This work is part of the UCF[5] project and partially funded by Systematic Paris Region Systems & ICT Cluster.

References

[1] Typescript. `https://www.typescriptlang.org/`.

[2] V. Balat. Ocsigen: typing web interaction with objective Caml. In *Proceedings of the ACM Workshop on ML, 2006*, pages 84–94, 2006.

[3] Benjamin Canou and Emmanuel Chailloux and Vincent Botbol. Static typing & JavaScript libraries: towards a more considerate relationship. In *22nd International World Wide Web Conference*, 2013.

[4] G. Berry. The Foundations of Esterel. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 425–454, 2000.

[5] G. Berry and M. Serrano. Hop and HipHop : Multitier Web Orchestration. *CoRR*, 2013.

[6] F. Boussinot and J.-F. Susini. The sugarCubes Tool Box: A Reactive Java Framework. *Softw. Pract. Exper.*, 28(14), Dec. 1998.

[7] D. Bunzli. React. `http://erratique.ch/logiciel/react`.

[8] E. Czaplicki. Blazing fast HTML. `http://elm-lang.org/blog/blazing-fast-html`, 2014.

[9] E. Czaplicki and S. Chong. Asynchronous Functional Reactive Programming for GUIs. PLDI '13, New York, NY, USA, 2013. ACM.

[10] R. El Sibaie and E. Chailloux. Pendulum : une extension réactive pour la programmation Web en OCaml. In *JFLA 2016 - Vingt-septièmes Journées Francophones des Langages Applicatifs*. (In French), Jan. 2016.

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

[12] Esterel Technologies. Scade Language Primer. Technical report, 2014.

[13] Facebook. Flow. `http://flowtype.org`, .

[14] Facebook. React. `http://facebook.github.io/react`, .

[15] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Springer-Verlag, Berlin, Heidelberg, 2010.

[16] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml System Release 4.03: Documentation And User's Manual. Technical report, Inria, 2015.

[17] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005*, 2005.

[18] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. *SIGPLAN Not.*, 44(10):1–20, Oct. 2009.

[19] D. Potop-Butucaru. *Compiling Esterel*. Springer, Berlin, 2007.

[20] M. Serrano. HOP: an environment for developing web 2.0 applications. In *International Lisp Conference, ILC 2007, Cambridge, Apr 1-4, 2007*, page 6, 2007.

[21] TasteJS. TodoMVC. `http://todomvc.com/`, 2014.

[22] J. Verbaten. Mercury, a truly modular frontend framework. `https://github.com/Raynos/mercury`, 2014.

[23] J. Vouillon and V. Balat. From bytecode to JavaScript: the Js_of_ocaml compiler. *Softw., Pract. Exper.*, 2014.

[24] W3C. Timing control for script-based animations. `http://www.w3.org/TR/animation-timing/`, 2015.

[5] `http://www.ubiquitus-content-framework.fr/`