Interactive Programming of Reactive Systems (Draft)

Louis Mandel Florence Plateau
LRI, Université Paris-Sud 11
LIENS, École Normale Supérieure
INRIA

November 2010

Abstract

The language ReactiveML is an extension of the general purpose programming language OCaml with synchronous reactive constructs. It is dedicated to program reactive systems such as video games or simulators. This paper presents `rmltop`, the ReactiveML counterpart of the OCaml toplevel. This toplevel allows a programmer to interactively write ReactiveML programs which are type-checked, compiled and loaded on the fly. The user can then progressively run concurrent processes and observe the interactions between them.

The main strength of `rmltop` is that all valid ReactiveML expressions are executed in the toplevel with the same semantics and efficiency as in the compiler. This allows to use the ReactiveML toplevel for design and debugging purposes. To illustrate the usefulness of this tool, we present some experiments on dynamic reconfiguration, done with the help of the toplevel. We finally describe an originality of `rmltop`: being itself a reactive system, it has been implemented in ReactiveML.

1 Introduction

ReactiveML is a programming language dedicated to the implementation of interactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the synchronous reactive model of Frédéric Boussinot [2] embedded in an ML language (here Objective Caml [12]). The synchronous reactive model provides synchronous parallel composition and dynamic features like dynamic creation of processes. ReactiveML is compiled into a purely sequential OCaml code. Native-code or bytecode executables are then generated by the OCaml compiler.

We propose here an interactive mode for ReactiveML in a way similar to the interaction loop (or toplevel) of OCaml. In this mode, ReactiveML programs can be defined and executed in an interactive manner. The toplevel (`rmltop`) reads ReactiveML phrases on the standard input, compiles them and executes them. Moreover it provides control directives to run a process, suspend the execution of the running processes, execute only the next n reactions or resume the execution. Those directives are directly launched in the toplevel. Additionally, the suspension directive can be launched by processes. It allows to program an *observer* that decides to suspend the execution when a certain condition is verified.

All these features make this execution mode a convenient tool for prototyping reactive behaviors. Contrary to sequential programs, reactive programs continuously interact with their environment. Modifying and programming the environment during the execution of the system is thus useful for testing and debugging. The ReactiveML toplevel is also useful to study dynamic reconfiguration of reactive programs. The addition of new processes during the execution can be used as a basic element to build a framework for programming reconfigurable applications [4, 8]. Finally, interacting with the processes behaviors improves the understanding of the reactive model. Hence the toplevel can help for teaching purposes.

The implementation of `rmltop` reuses the ReactiveML compiler. It has two consequences: (1) the implementation is small (about 500 source lines of code) and (2) the semantics of the two execution modes (the compiled mode and the interactive mode) are the same by construction. Moreover, a toplevel interpreter being itself a reactive system, one originality of `rmltop` is to be itself implemented

```
louis@machiavel> rmltop
        ReactiveML version 1.07.08
        Objective Caml version 3.11.2

# signal s;;
val s : ('_a, '_a list) event
val s : ('_a, '_a list) Implem.Lco_ctrl_tree_record.event = <abstr>
# let process p =
    await s;
    print_endline "Present";;
val p : unit process
val p : unit Implem.Lco_ctrl_tree_record.process = <fun>
# #run p;;
# emit s ();;
- : unit = ()
Present
```

Figure 1: An `rmltop` session.

in ReactiveML, making its implementation relatively elegant. `rmltop` is included in the distribution of ReactiveML which is available at: `http://rml.lri.fr`.

The ReactiveML toplevel is based on the original idea of Reactive Scripts [6] by Frédéric Boussinot and Laurent Hazard. Reactive Scripts is a scripting language first built on ReactiveC [5] and Tcl-Tk. Then Jean-Ferdy Susini proposed a new implementation [18] of this language based on SugarCubes [7] and Java. We will discuss the differences between our approach and Reactive Scripts in Section 5.

This paper is an extended version of [13]. In the following, we first present the use of `rmltop` in Section 2 through a collection of examples. Section 3 deals with reconfiguration aspects. Section 4 describes `rmltop` implementation. Section 5 is devoted to a comparison with related work and we conclude in Section 6.

2 Interactive Programming in ReactiveML

ReactiveML is based on the synchronous reactive model where time is defined as a succession of logical instants. In this model, the parallel composition guaranties that all processes can react at each instant and communications are made through instantaneous broadcasting of events. Hence, the language provides a way to define processes as functions that can be executed through several instants. The body of a process mixes OCaml code with reactive constructs *a la* Esterel [17]: parallel composition (`||` operator), emission of events (`emit`), etc. Moreover, as ReactiveML is an extension of OCaml,[1] we can also define data types and functions like in OCaml.

In the following, we assume that the reader has some notions of the OCaml language [12] and of synchronous programming [1]. We will not present the ReactiveML language in details, but only what is necessary for the following. A summary of ReactiveML constructs is available on annex A. For more details on ReactiveML the reader can refer to [14, 15].

2.1 First ReactiveML Interactive Session

Let us analyse the session given Figure 1. It is launched by the `rmltop` command and begins with the following line:

```
# signal s;;
val s : ('_a, '_a list) event
val s : ('_a, '_a list) Implem.Lco_ctrl_tree_record.event = <abstr>
```

This expression declares a global signal `s` (the character `#` is the prompt). It is followed by two pieces of information given by the toplevel: (1) the type inferred by the ReactiveML compiler for this declaration

[1]The current implementation of ReactiveML does not support objects, labels, polymorphic variants and functors.

and (2) the type of the corresponding OCaml code.[2]

Next, we define a process p (introduced by the keyword process) that prints Present when the signal s is emitted:

```
# let process p =
    await s;
    print_endline "Present";;
val p : unit process
```

Then an instance of p is executed by means of the #run directive (as in OCaml, directives begin with a # character):

```
# #run p;;
```

The instance of p now runs in background and the control is returned to the user.

Finally the signal s is emitted in the reactive machine:

```
# emit s ();;
```

Hence, the instance of the p process that is awaiting s reacts and the message Present is printed.

The main directive of rmltop is #run. This directive executes the process given as parameter. The directive #exec is derived from #run. It executes a reactive expression. It is a shortcut for:

$$\texttt{\#exec e} \equiv \texttt{\#run (process e)}$$

2.2 A Complete Example

We illustrate the use of rmltop on the so-called n-body problem. The n-body problem is the simulation of planets that obey the gravity laws of Newton. The entire code and a video presentation are available at http://rml.lri.fr/rmltop.

We first define the data type of planets, benefiting from the expressiveness of OCaml data types.

```
# type planet =
    { id : int;
      mass : float;
      pos : float * float * float;
      speed : float * float * float; } ;;
```

We now declare some useful constants and functions like the gravitational constant (g), the integration step (dt) and a function (random_speed) that creates a random speed value using the Random module of the OCaml standard library.

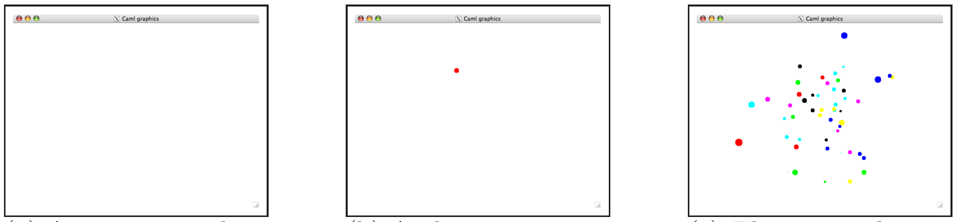
```
# let g = 6.67;;
val g : float
# let dt = 0.1;;
val dt : float

# let random_speed () =
    ((Random.float 100.0) -. 50.0,
     (Random.float 100.0) -. 50.0,
     (Random.float 100.0) -. 50.0) ;;
val random_speed : unit -> float * float * float
```

Some auxiliary OCaml functions are then defined but not detailed here.

A global signal env is declared. This signal will gather the position of all the planets in a list.

```
# signal env default [] gather (fun x y -> x::y);;
val env : ('_a, '_a list) event
```

[2] In the following, for the sake of conciseness, we do not always display the OCaml type output.



(a) An empty window (b) A planet (c) The sun and some other planets

Figure 2: Screenshots of the `window` process.

The gather function (here, `fun x y -> x::y`) defines how values emitted during the same instant are combined to build the value of the signal. This function adds each emitted value to the list of values that have already been emitted during the instant. This list is initialized with the default value `[]`.[3] The `env` signal has type `('_a, '_a list) event` where `'_a` will be instantiated with the type `planet`. It means that the values emitted on this signal must be of type `planet` and the value associated to the signal has type `planet list`.

The signal `env` is then used by the process `window` to display the planets.

```
# let update_display all = ... ;;
val update_display : planet list -> unit
# let process window =
    Graphics.open_graph "";
    Graphics.auto_synchronize false;
    loop
      await env (all) in  update_display all
    end ;;
val window : unit process
```

This process first initializes the graphical window and then enters into an infinite loop. The behavior of the loop is the following. The expression "`await env (all) in ...`" waits for the emission of the `env` signal, and binds `all` to the value associated to `env`. Then, at the instant following the emission of the signal, the body of `await/in` construct is executed. The `update_display` function uses the `all` value to draw all planets. Notice that there is no instantaneous loop since the `await/in` expression takes at least one instant.

This process is then executed by writing:

```
# #run window;;
```

Its effect is to open an empty OCaml graphics window (Fig 2(a)). Since the `env` signal is not emitted, the `window` process is stuck on its `await` expression.

Now the behavior of a planet is given by:

```
# let random_planet () = ... ;;
val random_planet : unit -> planet
# let compute_pos p all =  ... ;;
val compute_pos : planet -> planet list -> planet
# let process planet =
    let me = ref (random_planet()) in
    loop
      emit env !me;
      await env (all) in
      me := compute_pos !me all
    end ;;
val planet : unit process
```

[3]We could have written `signal env` which is a shortcut for `signal env default [] gather (fun x y -> x::y)`

The process `planet` uses two previously defined functions: (1) `random_planet` that creates a new planet at a random position with a random speed and (2) `compute_pos` that, given a planet `p` and a list of planets `all`, computes the new position of the planet `p` submitted to the attraction of all other planets.

The process `planet` creates a new random planet and enters in an infinite repetition of three parts. First it emits the position of the planet on the signal `env` to communicate it to other planets. Then it waits for the value of this signal (the list of all planets) which is available at the next instant. Finally, it uses this information to compute the new position of the planet.

To summarize, all the planets emit their position on the signal `env`. It is used by the `window` process for display and by each planet to compute its position at the next instant.

We now run an instance of the process `planet`:

```
# #run planet;;
```

This directive launches the `planet` process in background, in parallel with already running processes (here `window`). A new planet is created and appears on the graphical window (Figure 2(b)). As it is the unique body of the system, its trajectory is not modified and it goes out of the window. The `planet` process can be instantiated several times to create other planets.

```
# #run planet;;
# #run planet;;
```

As the planets are of non null weights, their trajectories are modified by the interaction with the other ones, but they still go out of the window. Nevertheless, as the value of the signal `env` is the list of the planets, we can verify that all created planets are still running by observing the value of `env`. To observe the value of the signal `env` in a stable state, we first use the `#suspend` directive. It asks for the suspension of the simulation at the beginning of the next instant.

```
# #suspend;;
```

Planets stop their movement. We can now observe the environment by typing:

```
# pre env;;
 - : bool = true
```

The evaluation of expression `pre env` returns the status (emitted or not) of the signal `env` at the preceding instant. `pre` can also be used to ask for the value of the signal:

```
# pre ?env;;
  - : planet list =
[{id = 1; mass = 1.;
  pos = (25486.668, -30490.001, -3332.7650);
  speed = (27.458270, -32.814312, -3.5462074)};
 {id = 2; mass = 1.;
  pos = (-17667.938, 18421.838, -12995.214);
  speed = (-34.721283, 35.976223, -25.585873)};
 {id = 3; mass = 1.;
  pos = (-5691.3338, 8876.0819, 6907.2694);
  speed = (-17.164963, 26.688011, 20.773542)}]
```

Observe that it is a list of three planets as expected and that their values at the preceding instant are displayed.

Let us now run a new planet.

```
# #run planet;;
```

As the simulation is currently suspended, we can use the `#step` directive that executes one instant of the system.

```
# #step;;
```

Displaying again the environment shows that a fourth planet has been added to the list, and that the other ones have moved:

```
# pre ?env;;
 - : planet list =
[{id = 1; mass = 1.;
  pos = (25492.159, -30496.564, -3333.4747);
  speed = (27.458270, -32.814312, -3.5462149)};
 {id = 2; mass = 1.;
  pos = (-17674.882, 18429.033, -13000.331);
  speed = (-34.721283, 35.976220, -25.585873)};
 {id = 3; mass = 1.;
  pos = (-5694.7668, 8881.4195, 6911.4241);
  speed = (-17.164961, 26.688011, 20.773546)};
 {id = 4; mass = 1.;
  pos = (-44.769087, -83.553279, -76.555396);
  speed = (12.309124, -15.532798, 12.909834)}]
```

The directives `#suspend` and `#step` are helpful for debugging and understanding reactive systems.[4] The directive `#resume` goes back to the sampled mode.

```
# #resume;;
```

We now define a new process `sun` that creates a star much heavier than the planets and which does not move:

```
# let process sun =
    let me =
      { id = 0;
        mass =  30000.0;
        pos = (0.0, 0.0, 0.0);
        speed = (0.0, 0.0, 0.0) }
    in
    loop
      emit env me;
      pause
    end ;;
val sun : unit process
```

Its behavior is to make its position available to the planets by emitting it at each instant on the `env` signal and to wait for the following instant without updating its position. If we run the `sun` process, a sun appears on the graphical window.

```
# #run sun;;
```

Several planets can be added at the same time using the `for/dopar` construct.

```
# #exec (for i = 1 to 50 dopar run planet done);;
```

The expression we execute here is a loop that runs 50 planets in parallel. Each new planet is attracted by the sun and turns around it (Figure. 2(c)).

Suppose we want to observe an eclipse.

```
# let eclipse { pos = (x, y, z) } =
    abs_float x < 10. && abs_float y < 10. && z > 0.;;
val eclipse : planet -> bool
```

The boolean function `eclipse` takes a planet as argument and tests if it is in front of the sun. We can test if at least one planet was at an eclipse position at the preceding instant by evaluating the following expression:

```
# List.exists eclipse (pre ?env);;
- : bool = false
```

[4]The directive `#step` n is also available. It allows to execute n instants of the system.

`List.exists p l` is a OCaml expression that returns `true` if an element of `l` satisfies the predicate `p`. Here, the returned value is `false`, so there was no eclipse when the phrase has been evaluated.

It would be very difficult to suspend by hand the simulation exactly when a planet is in front of the sun, and tedious to execute the system step by step until an eclipse occurs. Fortunately, the `#suspend` directive can be launched by processes. We can thus define a process `eclipse_observer` that suspends the simulation if a planet is at an eclipse position.

```
# let process eclipse_observer =
    loop
      await env (all) in
      if List.exists eclipse all then #suspend
    end;;
val eclipse_observer : unit process
# #run eclipse_observer;;
```

As soon as we run the eclipse observer, the simulation is suspended each time an eclipse occurs.

The process `eclipse_observer` is a *synchronous observer* [10]. This kind of processes observe dynamic properties of a system without modifying its behavior. The combination of this feature with the possibility to suspend the simulation allows to set semantic breakpoints. These breakpoints are defined by arbitrarily complex conditions expressed in the language itself. This is an original and powerful way to suspend the execution of a program.

We have shown through this example of the n-body problem that the ReactiveML toplevel is not only useful to understand a reactive system, but also to test and debug it. We are now going to describe how we can use the toplevel to do experiments about dynamic reconfiguration of reactive systems.

3 Dynamic Reconfiguration

We call dynamic reconfiguration the ability to add and remove processes to a system during its execution. The ReactiveML toplevel provides good basic elements to study dynamic reconfiguration of reactive systems. For example, it is easy to program a process `killable` that allows to kill each process *p* launched by the command `#run (killable` *p*`)` when its identifier is sent on a global signal `to_kill`.[5]

```
# signal to_kill ;;
val to_kill : (ident, ident list) event

# let process killable p =
    let id = gen_id () in print_id id;
    do
      let v = run p in Some v
    until to_kill(ids) when List.mem id ids -> None done ;;
val killable : 'a process -> 'a option process
```

The `killable` combinator is a higher order process: it takes a process as argument. It associates a fresh identifier to `p` using a `gen_id` function and prints it such that the user can know it. Then the body of the process executes `p` under the supervision of the global signal `to_kill`: the execution terminates when the identifier of the process belongs to the list of processes to kill (*i.e.* to the value associated to `to_kill`).

The behavior of the process `killable` is implemented using the ReactiveML `do` e_1 `until` s`(`x`)` `when` c `->` e_2 `done` construct. This construct executes e_1 until the signal s is present and c is satisfied, then it executes e_2 (in c and e_2, the variable x is bound to the value associated to the signal). Here, the condition `List.mem id ids` tests if the identifier of the process (`id`) belongs to the list of the processes to kill (`ids`). The value returned by the `killable` combinator is `None` if the execution of `p` is preempted. Otherwise, it is `Some v` where `v` is the value returned by `p`. Note that this combinator is polymorphic, it can be used to control the execution of a process that returns a value of any type.

[5] In the following, types may be specialized to simplify the reading. The type `ident` and the functions `gen_id` and `print_id` are supposed to be previously defined.

Killing a process launched through the `killable` combinator simply consists in emitting its identifier on the `to_kill` signal. Let us illustrate that on the example of previous section. We run a killable sun by typing:

```
# #run (killable sun);;
[1]
```

The identifier `1` of the process is displayed on the standard output. Then we put a planet into orbiter:

```
# #run planet;;
```

Finally, we remove the sun by emitting its identifier on the signal `to_kill`:

```
# emit to_kill 1;;
```

We can observe that the sun disappears and the planet goes out of the window because the sun is no more here to attract it.

3.1 Basic reconfiguration combinators

In this section, we present some simple combinators that provide a first step to reconfiguration. For the sake of conciseness, we consider that combinators take as argument processes identifiers. Under this hypothesis, the combinator `killable` becomes:

```
let process killable id p =
  do
    let v = run p in Some v
  until to_kill(ids) when List.mem id ids -> None done ;;
val killable : ident -> 'a process -> 'a option process
```

In the same way, it is possible to define a combinator that allows to suspend and resume a process (and only this one):

```
let process suspendable id p =
  control
    run p
  with to_suspend_resume(ids) when List.mem id ids done ;;
val suspendable : ident -> 'a process -> 'a process
```

The construct `control` e `with` $s(x)$ `when` c `done` switches between the execution of e and its suspension when the signal s is present and c is satisfied. Note that using the combinator `suspendable` differs from launching the `#suspend` directive. The former suspends one process, while the latter suspends the whole simulation. If we launch a process `sun` under the supervision of this combinator, it is possible to suspend it during a few instants in order to change the orbiter of the planets.

The `resettable` combinator allows to reset a process during its execution by the emission of its identifier on a global signal `to_reset`:

```
let rec process resettable id p =
  do
    run p
  until to_reset(ids) when List.mem id ids -> run (resettable id p) done ;;
val resettable : ident -> 'a process -> 'a process
```

Here, to reset the execution of the process `p`, it is first interrupted and then a new instantiation is executed. It is done through recursion (introduced with the `rec` keyword).

3.2 More advanced combinators

In this section, we illustrate how the behavior of a process can be changed dynamically, thanks to recursion and the ability to emit a process on a signal. A first useful reconfiguration combinator is the replacement of the behavior of a running process. The `replace` process runs an initial process `p_init` until the emission of a signal `new_behavior`. It then executes the process `p` carried by this signal. The process `p` is launched by a recursive call to `replace` to give the opportunity to replace it again.

```
let rec process replace p_init new_behavior =
  do
    run p_init; halt
  until new_behavior(p) -> run (replace p new_behavior) done ;;
val replace : 'a process -> ('b, 'a process) event -> 'c process
```

A design choice has been made here: the body of the `do/until` construct never dies (thanks to the `halt` statement). Thus, a process can be replaced even after its termination. If we remove the `halt` statement, the body of the `do/until` construct dies after the termination of the running process and thus the `replace` process also dies. In this case, a process can be replaced only before its death.

Now, we define a process `replaceable` similar to the combinators of previous section. It filters the replacement requests sent on a global signal `to_replace` and forwards to the process it manages the requests addressed to it.

```
signal to_replace ;;
val to_replace :
  (ident * unit process,  (ident * unit process) list) event

let process replaceable id p_init =
  signal new_behavior default process ()
                      gather (fun p q -> process (run p || run q))
  in
  run (replace p_init new_behavior)
  ||
  loop
    await to_replace(reqs) in
    List.iter (fun (x,p) -> if x = id then emit new_behavior p) reqs
  end ;;
val replaceable : ident -> unit process -> unit process
```

First, the process `replaceable` declares a local signal `new_behavior` dedicated to the collection of the requests concerning the managed process. The gathering function of this signal is such that if several requests are ordered at the same time, then the resulting new behavior will be the parallel composition of all requested new behaviors.
Then, the process `replaceable` is composed of two parallel branches. On the one hand, it launches the process `p_init` under the control of the `replace` process. On the second hand, it scans the global signal `to_replace` and forwards to `new_behavior` the requests concerning the process it manages.

Hence, launching the process *p* of identifier *id* with the `#run (replaceable` *id p*`)` directive allows to replace the behavior of *p* by the one of *p'* by emitting the pair `(`*id*`,` *p'*`)` on the global signal `to_replace`.

An other useful reconfiguration combinator is to add new behaviors to a running process. First, we define a `extend` process that executes a process `p_init` and awaits new processes to execute on a signal `new_behavior`.

```
let rec process extend p_init new_behavior =
  run p_init
  ||
  await new_behavior(p) in
  run (extend p new_behavior) ;;
val extend : 'a process -> ('b, 'a process) event -> unit process
```

Then, we have to program a process `extensible` that filters the adding requests send on a global signal `to_add`.

```
signal to_add ;;
val to_add : (ident * unit process,  (ident * unit process) list) event
```

The process `extensible` can be implemented similarly to the `replaceable` process. Hence, this code can be factorized by the introduction of a process `basic_requests_manager` parametrized by the reconfiguration combinator to apply (`combinator`) and the signal on which requests are sent (`requests`).

```
let process basic_requests_manager combinator requests id p_init =
  signal new_behavior default process ()
                      gather (fun p q -> process (run p || run q))
  in
  run (combinator p_init new_behavior)
  ||
  loop
    await requests(reqs) in
    List.iter (fun (x,p) -> if x = id then emit new_behavior p) reqs
  end ;;
val basic_requests_manager :
  (unit process -> ('a process, unit process) event -> 'b process) ->
  ('c, (ident * 'a process) list) event -> ident -> unit process ->
  unit process
```

With this manager, the processes `replaceable` and `extensible` are implemented as follows:

```
let replaceable = basic_requests_manager replace to_replace ;;
val replaceable : ident -> unit process -> unit process

let extensible = basic_requests_manager extend to_add ;;
val extensible : ident -> unit process -> unit process
```

A drawback of this implementation of these two combinators is that it does not allow to share information between the processes. In the case of `replaceable`, we would like to transmit a state from the running process to the new one. In the case of `extensible`, we would like the different behaviors to have a common state. This two goals can be achieved by writing a new manager. This new manager is very similar to `basic_requests_manager` except that the processes are parametrized by a state.

```
let process requests_manager combinator requests id p_init state =
  signal new_behavior default process ()
                      gather (fun p q -> process (run p || run q))
  in
  run (combinator (p_init state) new_behavior)
  ||
  loop
    await requests(reqs) in
    List.iter
      (fun (x,p) -> if x = id then emit new_behavior (p state))
      reqs
  end ;;
val requests_manager :
  ('a process -> (unit process, unit process) event -> unit process) ->
  ('b, (ident * ('state -> unit process)) list) event ->
  ident -> ('state -> 'a process) -> 'state -> unit process
```

An example illustrating the use of this manager is available at the address `http://rml.lri.fr/rmltop/reconfiguration`.

3.3 Composing combinators

We have seen that programming reconfiguration combinators in ReactiveML is natural, thanks to higher-order, polymorphism and recursion. An interesting point is that we are also able to combine them in order to produce more powerful ones.

Consider we want to launch a process in such a way that it is not only "suspendable", but also "killable". It can be done using the following combinator:

```
let killable_suspendable id p =
  killable id (suspendable id p) ;;
val killable_suspendable : ident -> 'a process -> 'a option process
```

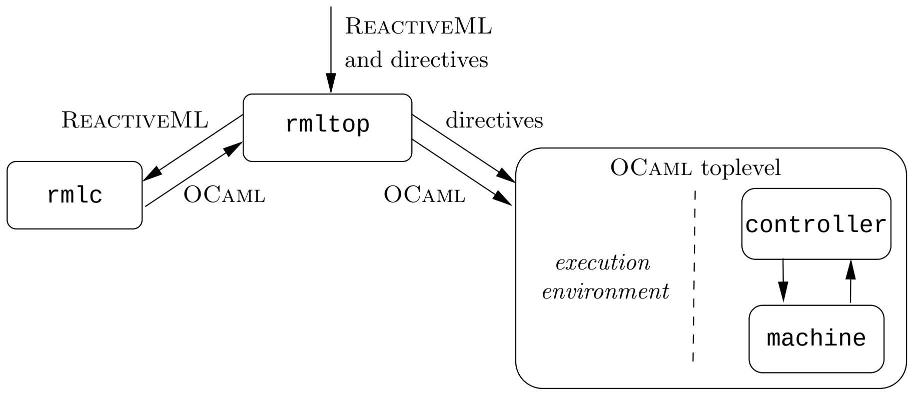


Figure 3: Structure of the implementation of the ReactiveML interactive mode.

This combinator allows to kill a process even while it is suspended. We could have done another design choice by composing the combinators differently:

```
let suspendable_killable id p =
  suspendable id (killable id p) ;;
val suspendable_killable : ident -> 'a process -> 'a option process
```

This later combinator does not allow to kill the process when it is suspended: the `suspendable` process controls the `killable` one, so it also suspends the behavior that allows to kill.

3.4 Conclusion

We have shown through these examples that it is easy to program reconfiguration combinators in ReactiveML. Trying to compose the different reconfiguration features raises many questions of design. What we argue here is that `rmltop` is a good tool to experiment the different choices and to understand the differences between them.

Finally, note that dynamic reconfiguration of reactive systems does not only consists in modifying behaviors but also in modifying data types. For this aspect of reconfiguration, ReactiveML does not provides any facilities. In particular, it is not possible to change the type of a signal during the execution. Nonetheless, this feature ensures the type safety of the system.

4 Implementation

The ReactiveML interactive mode consists of three parts: (1) a toplevel that reads the source code to build the execution environment and record the directives, (2) a reactive machine that evaluates processes launched by the `#run` directive and (3) a controller that supervises the execution of the reactive machine with respect to the other directives (`#suspend`, `#resume` and `#step`).

4.1 The Toplevel

As the ReactiveML language is compiled into OCaml, it is natural that the ReactiveML toplevel uses the OCaml toplevel to execute compiled ReactiveML phrases and to dynamically build an execution environment.

The structure of the implementation is presented Figure 3. A Unix process `rmltop` coordinates the parallel execution of a ReactiveML compiler `rmlc` and an OCaml toplevel `ocaml`.

- The ReactiveML compiler runs in an interactive mode. ReactiveML phrases are given as input to the `rmlc` compiler. The compiler returns the corresponding OCaml code.

- The compiled code is then sent to the OCaml toplevel that executes it. This execution builds an environment that contains the definition of the functions and processes.

The reactive machine and its controller run in a separate thread of the OCaml toplevel. This software architecture allows the reactive machine to access to the environment of the OCaml toplevel. The communication between the two threads is made through shared memory. So we use a lock `Rmltop_global.mutex` to prevent data races. This lock is taken during the execution of compiled ReactiveML phrases.

The execution of a directive sets a global reference in the execution environment. There is one reference by directive, defined in the module `Rmltop_global`. The controller inspect the values of the references to take into account the executed directives.

Let us now present the reactive machine.

4.2 The Reactive Machine

The reactive machine executes the processes launched by the `#run` directive. It is implemented by the function `Rmltop_reactive_machine.rml_react` that computes the reaction of one instant of the machine.

```
  val rml_react: unit Rmltop_global.rml_process list -> unit
```

It takes as argument a list of processes to execute in parallel with the processes that are already running in the machine. This function is the interface between the reactive machine and the controller.

The body of `rml_react` takes the lock `Rmltop_global.mutex` during its execution. It ensures that it is not executed in parallel with a ReactiveML phrase in the toplevel.

4.3 The Reactive Machine Controller

The controller makes the reactive machine react: it controls when the machine must compute a new instant. In particular, the controller interprets the following directives: `#suspend`, `#resume` and `#step`.

As shown in [11], the control of the execution of a reactive program is itself a reactive program. Thus, the control of the reactive machine can be programmed by a process written in ReactiveML. The core of the controller is a process `machine_controller`. It determines when the reactive machine must compute a new instant. It is composed of two modes: (1) the sampled mode and (2) the step by step mode. It must switch from the first one to the second one when the signal `suspend` is emitted, and from the second one to the first one when `resume` is emitted. When the machine is in the second mode, if the signal `step` is emitted, then a fix number of instants (given by the value associated to `step`) is computed. This computation can be interrupted if the signal `suspend` is emitted.

```
let process sampled =
  loop
    Rmltop_reactive_machine.rml_react(get_to_run());
    pause
  end

let process step_by_step =
  loop
    await step(n) in
    do
      for i = 1 to n do
        Rmltop_reactive_machine.rml_react(get_to_run());
        pause
      done
    until suspend done
  end

let process machine_controller =
  loop
    do run sampled until suspend done;
    do run step_by_step until resume done
  end
```

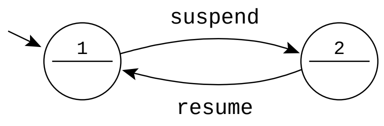


Figure 4: `machine_controller` automaton.

The process `machine_controller` implements the two states Moore automaton of Figure 4. We recall that the expression `do` e `until` s `done` executes its body (e) until signal s is present. The first `do/until` is the first state, and the second `do/until` is the second one. The condition to go from the first state to the second one is the presence of signal `suspend`. Respectively, the condition to go from the second state to the first one is the presence of signal `resume`. Let's now detail the code of each state.

In the sampled mode, an infinite loop periodically calls the reaction function of the machine.[6] In the step by step mode, each time the signal `step` is emitted with the value `n`, `n` instants of the reactive machine are executed. The `do/until` interrupts this sequence of reactions if the signal `suspend` is emitted.

The controller is also in charge of the translation of directives into ReactiveML signals. We have seen in Section 4.1 that the reactive machine and the controller communicate through shared variables (`suspend`, `resume` and `step`) defined in the `Rml_global` module. The controller monitors these global variables and emits the corresponding signal when the status of a variable changes. This behavior is implemented by the following `generate_signals` process.

```
let ref_to_sig ref s =
  match !ref with
  | None -> ()
  | Some v -> ref := None; emit s v

let process generate_signals =
  loop
    Mutex.lock Rmltop_global.global_mutex;
    ref_to_sig Rmltop_global.suspend suspend;
    ref_to_sig Rmltop_global.resume resume;
    ref_to_sig Rmltop_global.step step;
    Mutex.unlock Rmltop_global.global_mutex;
    pause;
  end
```

Finally, the behavior of the controller is to execute the two processes `machine_controller` and `generate_signals` in parallel.

```
let process controller =
  run machine_controller || run generate_signals
```

4.4 Conclusion

Due to the software architecture of `rmltop`, any valid ReactiveML expression is accepted by the toplevel and has the same semantics and efficiency as the compiled version (with a small overhead due to directive management). Indeed, the same ReactiveML compiler is used for the two versions of the language and the OCaml toplevel is as efficient as the compiler. Moreover this software architecture results in a light implementation.

5 Related Work

Though ReactiveML is largely inspired from Esterel, it would be much more difficult to implement a reactive toplevel for this language. In Esterel, adding dynamically processes to a running program

[6]The function `get_to_run` returns the list of processes to add to the machine and resets the list.

can introduce causality loops (logical inconsistency concerning presence and absence of signals). Indeed, the parallel composition of two causal expressions can be not causal. Here is an example:

```
signal s1, s2 in
present s1 then emit s2 else ()
|| present s2 then () else emit s1
```

According to ESTEREL semantics, if we suppose that `s1` is absent we can deduce that it is emitted in the *same* instant. If we suppose that it is present, we can deduce that it is not emitted. Hence, this program is absurd.

One strength of REACTIVEML is to rely on the Boussinot semantics that delays the reaction to absence. According to this semantics, if we suppose that `s1` is absent then we can deduce that it will be emitted at the *next* instant: there is no causality loop. Thanks to this feature, all programs are causal by construction. So, contrary to ESTEREL, it is always possible to add a process to a running machine.

REACTIVE SCRIPTS [6, 18] is a scripting language based on the reactive model and mixed with the Reactive Object Model [3]. It provides some powerful features like freezing an object such that it can be serialized and migrated to another reactive machine.

REACTIVE SCRIPTS is implemented as "macros" that are expanded into another reactive language (REACTIVEC or SUGARCUBES) which is then interpreted. Conversely, the REACTIVEML toplevel has a language approach in which programs are typed and compiled. It allows to define specific type systems and optimisations and provides a more efficient implementation. Another advantage of the REACTIVEML toplevel is that it accepts as input the whole REACTIVEML language, whereas REACTIVE SCRIPTS imposes some limitations on the host language.

ICOBJS [4, 8] is a graphical programming language also based on the reactive model of Boussinot. It was first implemented on top of REACTIVE SCRIPTS and is now directly implemented in JAVA. This language is built around the notion of `icobj` that can be seen as a basic behavior that can be killed, suspended, extended and serialized. A new `icobj` is built by the composition of the behavior of predefined ones.

Compared to the REACTIVEML toplevel, it is not possible to define dynamically new basic behaviors in ICOBJS. However, ICOBJS provides a powerful introspection mechanism that does not exist in REACTIVEML since it is not available in OCAML. One of the main differences is that ICOBJS is a complete framework where many design choices have been made whereas the REACTIVEML toplevel is much more flexible. It would be possible to implement a kind of ICOBJS in REACTIVEML by the composition of the combinators `killable`, `suspendable`, etc.

The REACTIVEML toplevel can also be compared to functional reactive programming (FRAN [9] and its successor FRP [19, 16]). FRAN and FRP are two programming languages embedded into HASKELL as libraries. Thus, they can be executed in an HASKELL toplevel.

In the context of interactive definition of reactive systems, the main difference comes from the programming model. FRP is a data flow language, whereas REACTIVEML is much more imperative. We believe that it is more natural to extend an imperative program with new parallel processes than to extend a data flow network. Indeed, the communication model of REACTIVEML is based on broadcasting and signals do not do any assumption on the number of events sources, thanks to their combination function.

6 Conclusion

We have presented `rmltop`, an interactive mode for the REACTIVEML language. It can be helpful to design and debug reactive systems and for teaching purposes. It provides a way to execute a program in a sampled mode or step by step and to dynamically modify the behavior of a system. An originality of this toplevel lies in the fact that it is itself coded in REACTIVEML. It results in a light and quite elegant code.

This experience has highlighted the usefulness of higher order and polymorphism of REACTIVEML for the definition of reconfiguration combinators. These combinators are meaningful beyond the scope of the toplevel: they can be useful in any REACTIVEML program.

Acknowledgements

We would like to thank Jean-Ferdy Susini for motivating us in taking advantage of the OCaml toplevel to implement `rmltop`, Marc Pouzet for his conclusive ideas of improvement and Frédéric Boussinot for its proofreading and advices.

References

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE, Special issue on embedded systems*, 91(1):64–83, January 2003.

[2] F. Boussinot and R. de Simone. The SL synchronous language. *Software Engineering*, 22(4):256–266, 1996.

[3] F. Boussinot, G. Doumenc, and J-B Stefani. Reactive objects. *Annales des Télécommunications*, 51(9-10):459–473, 1996.

[4] F. Boussinot, J-F. Susini, F. Dang Tran, and L. Hazard. A reactive behavior framework for dynamic virtual worlds. In *Proceedings of the sixth international conference on 3D Web technology*, 2001.

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

[6] Frédéric Boussinot and Laurent Hazard. Reactive scripts. In *Proceedings of the Third International Workshop on Real-Time Computing Systems Application (RTCSA'96)*, pages 267–274, 1996.

[7] Frédéric Boussinot and Jean-Ferdy Susini. The SugarCubes tool box : A reactive java framework. *Software Practice and Experience*, 28(4):1531–1550, 1998.

[8] Christian Brunette. *Construction et simulation graphiques de comportements: le modèle des Icobjs.* Thèse de doctorat, Université de Nice-Sophia Antipolis, 2004.

[9] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[10] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology*, pages 83–96, London, UK, 1994. Springer-Verlag.

[11] Grégoire Hamon and Marc Pouzet. Un simulateur synchrone pour Lucid Synchrone. In *Journées Francophones des Langages Applicatifs (JFLA'99)*, Morzine-Avoriaz, February 1999. INRIA.

[12] Xavier Leroy. The Objective Caml system release 3.10. Technical report, INRIA, 2007.

[13] Louis Mandel and Florence Plateau. Interactive programming of reactive systems. In *Proceedings of Model-driven High-level Programming of Embedded Systems (SLA++P'08)*, April 2008.

[14] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *Proceedings of 7th International conference on Principles and Practice of Declarative Programming (PPDP'05)*, 2005.

[15] Louis Mandel and Marc Pouzet. ReactiveML : un langage fonctionnel pour la programmation réactive. *Technique et Science Informatiques (TSI)*, 2007. Accepted for publication.

[16] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the Haskell Workshop*, September 2002.

[17] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel.* Springer, 2007.

[18] Jean-Ferdy Susini. *L'approche réactive au dessus de Java : sémantique et implémentation des SugarCubes et de Junior.* Thèse de doctorat, Ecole des Mines de Paris, 2001.

[19] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

A ReactiveML expressions

We present here the set of ReactiveML expressions that has been added to OCaml.

Terminal symbols are set in `typewriter font`. Non-terminal symbols are set in *italic font*. Square brackets *[]* denote optional components. Curly brackets *{}* denote zero, one or several repetitions of the enclosed components. Parentheses *()* denote grouping and */* denotes alternatives.

Process definitions

```
let process <id> { <pattern> } = <expr> in <expr>
process <expr>
```

Process definitions are introduced by the `process` keyword. A process can be named (`let process` *id*...) or anonymous (`process` *<expr>*).

Basic statements

```
nothing
pause
halt
run <process>
```

`nothing` is equivalent to `()`. `pause` suspends the execution until next instant. `halt` suspends the execution forever. `run` executes a process.

Compositions

```
<expr> ; <expr>
<expr> || <expr>
let <pattern> = <expr> { and <pattern> = <expr> } in <expr>
```

In ReactiveML, expressions can be composed in sequence or in parallel. The `let/and/in` construct computes several expressions in parallel and gets their values. Then it computes the body.

Signal declarations

```
signal <id> { , <id>  } in <expr>
signal <id> default <value> gather <function> in <expr>
```

These constructs declare new signals. When a signal is declared, we can define how to combine the values emitted during an instant with the `signal/gather` construct. If no combination function is given, the behavior of the signal is to collect all emitted values in a list.

Signal emission

```
emit <signal> [ <value> ]
```

Signal emissions are instantaneous broadcasting. Hence, a signal is present or absent during an instant but it cannot have both status. The notation `emit` *<signal>* is a shortcut for `emit` *<signal> ()*.

Signal status

```
present <signal> then <expr> else <expr>
await [ immediate ] <signal>
pre <signal>
```

The expression `present` tests the status of a signal. If the signal is present, the `then` branch is executed instantaneously, otherwise the `else` branch is executed at the following instant.

The expression `await s` waits `s` to be emitted and terminates at the following instant. Whereas the expression `await immediate s` waits `s` to be emitted and terminates instantaneously.

Like in ESTEREL, the non-immediate version of `await` is the default one such that `await s; await s` waits two occurrences of `s`, while the expression `await immediate s; await immediate s` is equivalent to `await immediate s`.

The expression `pre s` evaluates to `true` if the signal `s` has been emitted at the preceding instant. Otherwise, it evaluates to `false`.

Signal value

```
await <signal> (<pattern>) [ when expr ] in <expr>
await [ immediate ] one <signal> (<variable>) in <expr>
pre ?<signal>
last ?<signal>
default ?<signal>
```

The `await/in` waits the emission of a signal. At the instant following the emission, the body is executed in an environment where the pattern is bind to the value of the signal (the combination of the values emitted at the preceding instant). Notice that the `await/in` keeps waiting when the value of the signal does not match the pattern or if the condition specified after the `when` keyword is not satisfied.

The `await/one/in` construct waits the emission of a signal to bind the pattern with one of the emitted values. In case of multiple emission during an instant, the choice of the value is not specified. Like `await`, the body of the expression is executed at the instant following the reception of the signal (except if there is the `immediate` keyword). To be causal by construction, there is no immediate version of the `await/in` construct.

The expression `pre ?s` evaluates to the value associated to `s` at the preceding instant. If `s` has not been emitted at the preceding instant, `pre ?s` is equal to the default value given at the declaration point of the signal. `last ?s` has a slight different behavior. It evaluates to the last value associated to `s` when it was emitted. While `s` has never been emitted, `pre ?s` and `last ?s` both evaluates to the default value of `s`.

The `default` function returns the default value of a signal.

Iterators

```
loop <expr> end
while <expr> do <expr> done
for <id> = <expr> ( to | downto ) <expr> do <expr> done
for <id> = <expr> ( to | downto ) <expr> dopar <expr> done
```

`loop` is an infinite loop. `while` and `for/do` are the classical loops. They execute their body several times in sequence. Contrarily, the `for/dopar` loop executes its body several times in parallel.

Control structures

```
do <expr> when <signal> done
control <expr> with <signal> [ (<pattern>) [ when <expr> ] ] done
do <expr>
until <signal> [ (<pattern>) [ when <expr> ] ] [ -> <expr> ] done
```

`do/when` and `control/with` allows to suspend the execution of an expression. The `do/when` executes its body only when the signal is present. The `control/with` switches between an active mode and a suspended one each time that the signal is present.

The preemption construct `do/until` stops the execution of its body at the end of instant when the signal is emitted. An handler executed in case of preemption can be associated to this construct.

As in the `await/in` construct, additional constraints can be put on the suspension and preemption, through the pattern and `when` clause that impose conditions on the value of the signal.