

Electronic Notes in Theoretical Computer Science 238 (2009) 21–36

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Interactive Programming of Reactive Systems

Louis Mandel[1](#_bookmark0) Florence Plateau[2](#_bookmark0)

Laboratoire de Recherche en Informatique Universite´ Paris-Sud 11, CNRS*, Orsay F-91405* Inria Saclay Ile-de-France*, Orsay F-91893*

**Abstract**

ReactiveML is a synchronous reactive extension of the general purpose programming language OCaml. It allows 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 accepted in the toplevel with the same semantics as in the compiler. This allows to use the ReactiveML toplevel as a debugger. Furthermore, the interpreted code is as efficient as if it was compiled. Moreover, a toplevel interpreter being itself a reactive system, another originality of rmltop is its own implementation in ReactiveML which makes it relatively light.

*Keywords:* Reactive Programming, Prototyping, Debugging, Reactive Scripts.

# Introduction

ReactiveML is a programming language dedicated to the implementation of in- teractive systems as found in graphical user interfaces, video games or simulation

problems. ReactiveML is based on the synchronous reactive model of Fr´ed´eric

Boussinot [[3](#_bookmark16)] embedded in an ML language (here OCaml [[14](#_bookmark27)]). The synchronous

reactive model provides synchronous parallel composition and dynamic features like dynamic creation of processes. ReactiveML is compiled into a purely sequen- tial 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

1 Email: [louis.mandel@lri.fr](mailto:louis.mandel@lri.fr)

2 Email: [florence.plateau@lri.fr](mailto:florence.plateau@lri.fr)

1571-0661/© 2009 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2008.01.004

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 re- active behaviors. Contrary to sequential programs, reactive programs continuously interact with their environment. Modifying and programming the environment dur- ing the execution of the system is thus useful for testing and debugging. It can also help for teaching purposes: the interaction with the processes behaviors improves the understanding of the reactive model.

The ReactiveML toplevel is also useful to study dynamic reconfiguration of re-

active programs. The addition of new processes during the execution can be used as a basic element to build a framework for programming reconfigurable applications. Icobjs [[5](#_bookmark18),[10](#_bookmark22)], a graphical programming language based on the reactive model, is

an example of such a framework.

rmltop is included in the distribution of ReactiveML which is available at:

[http://rml.inria.fr](http://rml.inria.fr/).

Any ReactiveML program accepted by the compiler can be executed in the toplevel. Moreover, the execution is as efficient as the compiled version of the program. The ReactiveML toplevel does not interpret programs. It compiles them into bytecode and executes the bytecode.

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 (compiled and interactive) are the same by construction. Moreover, a toplevel interpreter being itself a reactive system, one originality of rmltop is to be itself implemented in ReactiveML, making its implementation relatively elegant.

The

ReactiveML

toplevel is based on the original idea of

Reactive

Scripts [[8](#_bookmark21)] by Fr´ed´eric Boussinot and Laurent Hazard. Reactive Scripts is a

scripting language first built on ReactiveC [[6](#_bookmark19)] and Tcl-Tk. Then Jean-Ferdy

Susini proposed a new implementation [[18](#_bookmark31)] of this language based on

Sugar-

Cubes [[9](#_bookmark23)] and Java. We will discuss the differences between our approach and

Reactive Scripts in Section [4.1](#_bookmark11).

In the following, we first present the use of rmltop in Section [2](#_bookmark1) through a collection of examples. Section [3](#_bookmark6) describes its implementation. Section [4](#_bookmark12) is devoted to a discussion and we conclude in Section [5](#_bookmark13).

# Interactive Programming in ReactiveML

ReactiveML is an extension of OCaml [3](#_bookmark2) such that we can define data types and

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

louis@machiavel> rmltop

ReactiveML version 1.06.07 Objective Caml version 3.10.1

# 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

Fig. 1. An rmltop session.

functions like in OCaml. Moreover, it provides synchronous reactive processes as functions that can be executed through several instants.

The body of a process mixes OCaml code with reactive constructs *`a la* Esterel. Programs can be composed in parallel (|| operator) and communication is based on broadcast with associated construct to emit signals (emit), test their presence (present, await), etc.

Notice that contrary to Esterel, ReactiveML follows the Boussinot seman- tics: the reaction to absence of signals is delayed. This restriction ensures that programs are causal by construction (a signal cannot be present and absent during an instant). In the following, we assume that the reader has some notions of the OCaml language [[14](#_bookmark27)] and of synchronous programming [[1](#_bookmark14)]. We will not present the ReactiveML language in details here, but only what is necessary for the following. For more details on ReactiveML the reader can refer to [[15](#_bookmark28),[16](#_bookmark29)].

* 1. *First* ReactiveML *Session*

An rmltop session is a succession of definitions (types, values, processes ...), exe- cutions of directives and ReactiveML instantaneous expressions. An example of session is given Fig. [1](#_bookmark3). It is launched by the rmltop command.

The session begins with the following line:

# signal s;;

that declares a global signal s (the character # is the prompt). This declaration is followed by two pieces of information given by the toplevel: (1) the type in- ferred by the ReactiveML compiler for this declaration and (2) the type of the corresponding OCaml code.

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";;

Then an instance of p is executed by means of the #run directive (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 implemented as follows: #exec e;; *≡* #run (process e);;

* 1. *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.inria.fr/slap08>.

We first define the data type of planets.

# type planet =

{ id : int; mass : float;

pos : float \* float \* float;

speed : float \* float \* float; } ;;

We can notice here that we benefit from the expressiveness of OCaml data types. We now declare some useful constants and functions like the gravitational con- stant, the integration step and a function that creates a random speed value. The

random\_speed function uses the Random module of the OCaml standard library.

# let g = 6.67;;

*val g : float*

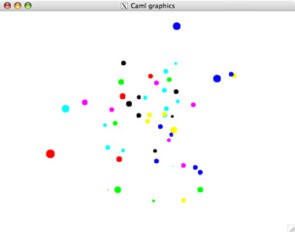
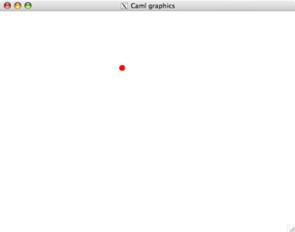
*val g : float = 6.67*

# let dt = 0.1;;

*val dt : float*

*val dt : float = 0.1*

#let random\_speed () =



(a) An empty window (b) A planet (c) The sun and some planets Fig. 2. Screenshots of the window process.

|  |  |  |  |
| --- | --- | --- | --- |
| ((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*

*val random\_speed : unit -> float \* float \* float = <fun>*

Other pure OCaml functions are then defined but not detailed here. A global signal

env is declared. This signal will gather the positions of all the planets in a list.

# signal env;;

*val env : (’\_a, ’\_a list) event*

*val env : (’\_a, ’\_a list) Implem.Lco\_ctrl\_tree\_record.event = <abstr>*

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 for display by process window.

# let process window = Graphics.open\_graph ""; Graphics.auto\_synchronize false; loop

await env (all) in update\_display all end ;;

*val window : unit process*

*val window : unit Implem.Lco\_ctrl\_tree\_record.process = <fun>*

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 func- tion 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)](#_bookmark4)). 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*

*val random\_planet : unit -> planet = <fun>*

# let compute\_pos p all = ... ;;

*val compute\_pos : planet -> planet list -> planet*

*val compute\_pos : planet -> planet list -> planet = <fun>*

# 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*

*val planet : unit Implem.Lco\_ctrl\_tree\_record.process = <fun>*

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 rep- etition 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 the process planet.

# #run planet;;

It launches the planet process in background, in parallel with already running processes (here window). It creates a new planet that appears on the graphical window (Fig. [2(b)](#_bookmark5)). As it is the unique body of the system, its trajectory is not modified and it goes out of the window. We can run several times the planet process 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 using the Re- activeML pre operator.

# pre env;;

*- : bool = true*

The evaluation of expression pre env returns the status (emitted or not) of 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)}]*

We can observe that it is indeed a list of three planets, and we can consult the values of the different fields. Note that it is not possible to ask for the status or the value of a signal at the current instant: it’s an intrinsic feature of the reactive model.

When the execution is suspended, we can call another useful directive: the #step directive. It executes one instant of the system.

# #run planet;; # #step;;

# 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)}]*

Here we run a new planet, then we execute one step of the system and we ask for the env value. We can notice that a fourth planet has indeed been added to the list, and the other ones have moved.

The directives #suspend and #step are helpful for debugging and understanding reactive systems. Notice that the directive #step *n* is also available. It allows to execute *n* instants of the system.

The directive #resume goes back to the sampled mode.

# #resume;;

We now define a new process sun that creates a planet much heavier than the other ones 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*

*val sun : unit Implem.Lco\_ctrl\_tree\_record.process = <fun>*

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.

If we run the sun process, a sun appears on the graphical window.

# #run sun;;

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

To add several planets at the same time, we use the primitive #exec that allows to launch a non-instantaneous ReactiveML expression. The expression we execute here is a loop that runs 50 planets in parallel (for/dopar construct). We can observe that each newly added planet is attracted by the sun and turns around it (Fig. [2(c)](#_bookmark5)).

We now want to observe eclipses.

# let eclipse { pos = (x, y, z) } =

abs\_float x < 10. && abs\_float y < 10. && z > 0.;;

*val eclipse : planet -> bool*

*val eclipse : planet -> bool = <fun>*

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 a planet was at an eclipse position at the preceding instant by evaluating the following expression:

# List.exists eclipse (pre ?env);;

*- : bool = false*

It applies the exist function of the List module of OCaml to the planets envi- ronment. 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*

*val eclipse\_observer : unit Implem.Lco\_ctrl\_tree\_record.process = <fun>*

# #run eclipse\_observer;;

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

The process eclipse\_observer is a synchronous observer. It can observe dy- namic properties without modifying the behavior of the system. 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 ex- pressed 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 Reac- tiveML toplevel is not only useful to understand a reactive system, but also to test and debug it. We are now going to describe its implementation.

# 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,

1. a reactive machine that evaluates processes launched by the #run directive and
2. a controller that supervises the execution of the reactive machine with respect to the other directives (#suspend, #resume and #step).
   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 dynamically builds an execution environment.



Rml

+ Directives

Rml

rmltop

rmlc

Ocaml

Ocaml ocaml toplevel

+ Directives

controller

*execution environment*

machine

Fig. 3. Structure of the implementation of the ReactiveML interactive mode.

The structure of the implementation is presented Fig. [3](#_bookmark8). 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 share the en- vironment of the OCaml toplevel. The communication between the two threads is made through shared memory. So we use a lock Rmltop\_global.mutex to pre- vent data races. This lock is taken during the execution of ReactiveML phrases (compiled into OCaml).

The execution of a directive sets a global reference in the execution environment. The controller can then access the reference to treat the directive. There is one reference by directive, defined in the module Rmltop\_global. Let us now present the reactive machine.

* 1. *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 com- putes 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.

suspend



1

2

resume

Fig. 4. machine controller automaton.

The body of rml\_react takes the lock Rmltop\_global.mutex during its execu- tion. It ensures that it is not executed in parallel with a ReactiveML phrase in the toplevel.

* 1. *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 [[12](#_bookmark24)], the control of the execution of a reactive program is itself a re- active 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

The process machine\_controller implements the two states Moore automaton of

Fig. [4](#_bookmark9). 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. [4](#_bookmark10) 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 [3.1](#_bookmark7) 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 imple- mented 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

* 1. *Conclusion*

Due to the software architecture of rmltop, any valid ReactiveML expression is accepted in the toplevel and has the same semantics and efficiency as the compiled version. Indeed, the same ReactiveML compiler is used for the two versions of the language and the OCaml toplevel is as efficient as the bytecode compiler. Moreover this software architecture results in a light implementation.

4 The function get to run returns the list of processes to add to the machine and resets the list.

# Discussions

* 1. *Related Work*

First, we can remark that it is not possible to implement a reactive toplevel based on the semantics of Esterel. In this model, processes cannot be dynamically added to a running program because causality loops may appear when two expressions are composed in parallel. For example, in the following expression, even if each present expression is causal, the parallel composition of the two is not causal.

signal s1, s2 in

present s1 then emit s2 else ()

|| present s2 then () else emit s1

Here, with 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.

With ReactiveML semantics, if we suppose that s1 is absent then we can de- duce that it will be emitted at the *next* instant: there is no causality loop. With the reactive model of Boussinot, all programs are causal by construction. So, contrary to Esterel, it is always possible to add a process to a running machine.

Reactive Scripts [[8](#_bookmark21),[18](#_bookmark31)] is a scripting language based on the reactive model and mixed with the Reactive Object Model [[4](#_bookmark17)]. It provides some powerful features like freezing an object such that it can be serialized and migrate to another reactive machine.

Reactive Scripts is implemented as “macros” that are expanded into another reactive language (ReactiveC or SugarCubes) which is then interpreted. Con- versely, ReactiveML 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 is less expressive than ReactiveC and SugarCubes. It imposes some limitations on the host language.

* 1. *Dynamic Reconﬁguration*

The combination of the reactive model with the ability to dynamically define and add new processes provides good basic elements to study dynamic reconfiguration of reactive systems. For example, it is easy to extend the ReactiveML toplevel such that each process *p* launched by the #run (killable *p*) command can be killed by the emission of its identifier on a kill signal.

# let process killable p =

let id = gen\_id () in print\_endline ("["^(string\_of\_int id)^"]"); signal kill\_me in

do

run p; emit kill\_me

|| loop

await kill(ids) in if List.mem id ids then emit kill\_me end

until kill\_me done;;

*val killable : ’a process -> unit process*

The killable process is a higher order process. It associates a fresh identifier to p using the 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 a kill\_me signal: the presence of this signal interrupts the execution. kill\_me is emitted when the identifier of the process belongs to the list of processes to kill (the value associated to kill) or when the execution of p is terminated.

In a same way, it is for example possible to define a kind of icobj combinator that automatically provides the possibility to suspend and resume a process (and only this one), to add a process inside a running icobj, etc. A strength of ReactiveML is polymorphism and higher order that allow to easily program such combinators.

Note that dynamic reconfiguration of reactive systems does not only consist in modifying the behavior 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 during the execution the type of a signal but this feature ensures the type safety of the system.

* 1. *Language Extension*

The implementation of rmltop has highlighted an extension of the ReactiveML

language that should be interesting to consider.

To implement the communication between the OCaml toplevel and the con- troller we had to use shared memory and a mutex. Moreover, the process generate\_signals had to do active waiting. Thus it would be interesting to have asynchronous tasks.

We are currently working on an extension of ReactiveML with asynchronous

concurrent constructs based on the join-calculus [[11](#_bookmark25)] similar to the ones of

Jo-

Caml

[[13](#_bookmark26)]. With this extension, a function new\_cell that creates a one place

buffer could be written as follows:

let new\_cell () =

def state (\_) & set(x) = state(Some x) & reply () to set

or state (Some x) & get() = state(None) & reply x to get in spawn (state None);

(set, get)

*val new\_cell : (’a -> unit process, unit -> ’a process)*

The body of the function contains a join-definition (def/in) that introduces three channels (state, set and get). This join-definition is made of two reaction rules. The first one defines the behavior of set: it updates the state of the buffer. The second one defines the behavior of get: it returns the value contained in the buffer.

Notice that a call to get is blocked until the value on the state channel matches the pattern Some x. The expression spawn (state None) initializes the state of the buffer.

For example, this buffer can be used to communicate the value of #step directive as follows:

let set\_step, get\_step = new\_cell() let process generate\_step =

loop let n = run (get\_step ()) in emit step n ; pause end

The work on this extension is related with some other language like Loft [[7](#_bookmark20)],

Fair Threads [[17](#_bookmark30)] or ULM [[2](#_bookmark15)].

# 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 elegant code, that could be even better if ReactiveML would provide asynchronous tasks. This gives a good motivation to add such a feature to the ReactiveML language.

# Acknowledgement

We would like to thank Jean-Ferdy Susini for motivating us in taking advantage of the OCaml toplevel to implement rmltop, and Marc Pouzet for his conclusive ideas of improvement. We also thank the referees for their useful comments.

# 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. G´erard Boudol. ULM: A core programming model for global computing. In *Proceedings of the 13th European Symposium on Programming (ESOP’2004)*, pages 234–248, 2004.
3. F. Boussinot and R. de Simone. The SL synchronous language. *Software Engineering*, 22(4):256–266, 1996.
4. F. Boussinot, G. Doumenc, and J-B Stefani. Reactive objects. *Annales des T´el´ecommunications*, 51(9-10):459–473, 1996.
5. 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.
6. Fr´ed´eric Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, April 1991.
7. Fr´ed´eric Boussinot. Concurrent programming with Fair Threads: The LOFT language, 2003.
8. Fr´ed´eric 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.
9. Fr´ed´eric Boussinot and Jean-Ferdy Susini. The SugarCubes tool box : A reactive java framework.

*Software Practice and Experience*, 28(4):1531–1550, 1998.

1. Christian Brunette. *Construction et simulation graphiques de comportements: le mod`ele des Icobjs*. Th`ese de doctorat, Universit´e de Nice-Sophia Antipolis, 2004.
2. C´edric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of Principles of programming languages (POPL’96)*, pages 372–385. ACM Press, 1996.
3. Gr´egoire Hamon and Marc Pouzet. Un simulateur synchrone pour Lucid Synchrone. In *Journ´ees* *Francophones des Langages Applicatifs (JFLA’99)*, Morzine-Avoriaz, February 1999. INRIA.
4. JoCaml. [http://jocaml.inria.fr](http://jocaml.inria.fr/).
5. Xavier Leroy. The Objective Caml system release 3.10. Technical report, INRIA, 2007.
6. 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.
7. Louis Mandel and Marc Pouzet. ReactiveML : un langage fonctionnel pour la programmation r´eactive.

*Technique et Science Informatiques (TSI)*, 2007. Accepted for publication.

1. Manuel Serrano, Fr´ed´eric Boussinot, and Bernard Serpette. Scheme Fair Threads. In *Proceedings of 6th International conference on Principles and Practice of Declarative Programming (PPDP’04)*, 2004.
2. Jean-Ferdy Susini. *L’approche r´eactive au dessus de Java : s´emantique et impl´ementation des SugarCubes et de Junior*. Th`ese de doctorat, Ecole des Mines de Paris, 2001.