```
(**************************************************************************)
(*                                                                        *)
(*                           ReactiveML                                   *)
(*                    http://reactiveML.org                               *)
(*                    http://rml.inria.fr                                 *)
(*                                                                        *)
(*                         Louis Mandel                                   *)
(*                                                                        *)
(*  Copyright 2002, 2007 Louis Mandel.  All rights reserved.              *)
(*  This file is distributed under the terms of the GNU Library           *)
(*  General Public License, with the special exception on linking         *)
(*  described in file ../LICENSE.                                         *)
(*                                                                        *)
(*  ReactiveML has been done in the following labs:                       *)
(*  - theme SPI, Laboratoire d'Informatique de Paris 6 (2002-2005)        *)
(*  - Verimag, CNRS Grenoble (2005-2006)                                  *)
(*  - projet Moscova, INRIA Rocquencourt (2006-2007)                      *)
(*                                                                        *)
(**************************************************************************)

module Lk_interpreter: Lk_interpreter.S  =
  functor (Event: Sig_env.S) ->
  struct

    exception RML

    type ('a, 'b) event =
        ('a,'b) Event.t * waiting ref * waiting ref

    and control_tree =
        { kind: control_type;
          mutable alive: bool;
          mutable susp: bool;
          mutable cond: (unit -> bool);
          mutable children: control_tree list;
          mutable next: next; }
    and control_type =
        Top
      | Kill of (unit -> unit step)
      | Susp
      | When of unit step ref

    and 'a step = 'a -> unit
    and next = unit step list
    and current = unit step list
    and waiting = unit step list
    and 'a process = 'a step -> control_tree -> unit step
    and join_point = int ref


(* list of processes to execute in the current instant *)
    let current = ref ([]: current)

(* End of instant flag *)
    let eoi = ref false

(* End of instant waiting list *)
    let weoi = ref ([]: waiting)

(* Dummy step function *)
    let dummy_step _ = ()


(* wake up processes waiting on an event *)
    let wakeUp w =
      current := List.rev_append !w !current;
      w := []

(* list of lists of processes to wake up at the end of instant *)
```



```
    let toWakeUp = ref []
    let wakeUpAll () =
      List.iter (fun wp -> wakeUp wp) !toWakeUp;
      toWakeUp := []


(* root of the control tree *)
    let top =
      { kind = Top;
        alive = true;
        susp = false;
        children = [];
        cond = (fun () -> false);
        next = []; }


(* compute the new state of the control tree *)
(* and move in the list [current] the processes that are *)
(* in the [next] lists. *)
    let eval_control_and_next_to_current =
      let rec eval pere p active =
        if p.alive then
          match p.kind with
          | Top -> raise RML
          | Kill handler ->
              if p.cond()
              then
                (pere.next <- (handler()) :: pere.next;
                 false)
              else
                (p.children <- eval_children p p.children active [];
                 if active then next_to_current p
                 else next_to_father pere p;
                 true)
          | Susp ->
              let pre_susp = p.susp in
              if p.cond() then p.susp <- not pre_susp;
              let active = active && not p.susp in
              if pre_susp
              then
                (if active then next_to_current p;
                 true)
              else
                (p.children <- eval_children p p.children active [];
                 if active then next_to_current p
                 else if not p.susp then next_to_father pere p;
                 true)
          | When f_when ->
              if p.susp
              then true
              else
                (p.susp <- true;
                 pere.next <- !f_when :: pere.next;
                 p.children <- eval_children p p.children false [];
                 true)
        else
          false

      and eval_children p nodes active acc =
        match nodes with
        | [] -> acc
        | node :: nodes ->
            if eval p node active
            then eval_children p nodes active (node :: acc)
            else eval_children p nodes active acc

      and next_to_current node =
        current := List.rev_append node.next !current;
        node.next <- []
```

```
      and next_to_father pere node =
        pere.next <- List.rev_append node.next pere.next;
        node.next <- []
      in
      fun () ->
        top.children <- eval_children top top.children true [];
        next_to_current top

(* Move in the [current] list the processes that are in the [next] lists *)
    let rec next_to_current p =
      if p.alive && not p.susp then
        (current := List.rev_append p.next !current;
         p.next <- [];
         List.iter next_to_current p.children)
      else ()

(* Create new events *)
    let new_evt_combine default combine =
      (Event.create default combine, ref [], ref [])

    let new_evt_memory_combine default combine =
      (Event.create_memory default combine, ref [], ref [])

    let new_evt() =
      new_evt_combine [] (fun x y -> x :: y)


(* ------------------------------------------------------------------------ *)
    let sched () =
      match !current with
      | f :: c ->
          current := c;
          f ()
      | [] -> ()

(* ------------------------------------------------------------------------ *)
    let rml_pre_status (n, _, _) = Event.pre_status n

    let rml_pre_value (n, _, _) = Event.pre_value n

    let rml_last (n, _, _) = Event.last n

    let rml_default (n, _, _) = Event.default n


(* ------------------------------------------------------------------------ *)
    let rml_global_signal = new_evt

    let rml_global_signal_combine = new_evt_combine

    let rml_global_signal_memory_combine = new_evt_memory_combine

(* ------------------------------------------------------------------------ *)
(*************************************)
(* compute                           *)
(*************************************)

    let rml_compute_v v k _ =
      k v

    let rml_compute e k _ =
      k (e())

(*************************************)
(* pause                             *)
(*************************************)
    let rml_pause k ctrl _ =
      ctrl.next <- k :: ctrl.next;
      sched ()
```



```
(*************************************)
(* pause_kboi                        *)
(*************************************)
    let rml_pause_kboi k ctrl _ =
      raise RML

(*************************************)
(* halt                              *)
(*************************************)
    let rml_halt _ =
      sched ()

(*************************************)
(* halt_kboi                         *)
(*************************************)
    let rml_halt_kboi _ =
      sched ()

(*************************************)
(* emit                              *)
(*************************************)
    let set_emit (n,wa,wp) v =
      Event.emit n v;
      wakeUp wa;
      wakeUp wp

    let rml_emit_v_v v1 v2 k _ =
      set_emit v1 v2;
      k ()

    let rml_emit_v_e v1 e2 k _ =
      rml_emit_v_v v1 (e2()) k ()

    let rml_emit_e_v e1 v2 k _ =
      rml_emit_v_v (e1()) v2 k ()

    let rml_emit_e_e e1 e2 k _ =
      rml_emit_v_v (e1()) (e2()) k ()

    let rml_emit = rml_emit_e_e

    let rml_emit_pure_v v1 =
      rml_emit_v_v v1 ()

    let rml_emit_pure_e e1 =
      rml_emit_v_v (e1()) ()

    let rml_emit_pure = rml_emit_pure_e

    let rml_expr_emit s v =
      rml_emit_v_v s v (fun () -> ()) ()

    let rml_expr_emit_pure evt = rml_expr_emit evt ()


(*************************************)
(* present                           *)
(*************************************)
    let rml_present_v (n,_,wp) k_1 k_2 =
      let rec self = fun _ ->
        if Event.status n
        then
          k_1 ()
        else
          if !eoi
          then
            (ctrl.next <- k_2 :: ctrl.next;
             sched ())
```

```
          else
            (wp := self :: !wp;
             toWakeUp := wp :: !toWakeUp;
             sched ())
      in
      fun _ -> self ()

    let rml_present ctrl expr_evt k_1 k_2 _ =
      let evt = expr_evt () in
      rml_present_v ctrl evt k_1 k_2 ()


(*************************************)
(* await_immediate                   *)
(*************************************)

    let rml_await_immediate_v evt k ctrl _ =
      let rec self _ =
        rml_present_v ctrl evt k self ()
      in
      self ()

    let rml_await_immediate expr_evt k ctrl _ =
      let evt = expr_evt() in
      rml_await_immediate_v evt k ctrl ()

(*************************************)
(* get                               *)
(*************************************)
    let step_get_eoi n f ctrl _ =
      let v =
        if Event.status n
        then Event.value n
        else Event.default n
      in
      ctrl.next <- (f v) :: ctrl.next;
      sched()

    let rml_get_v (n,_,_) f ctrl _ =
      weoi := (step_get_eoi n f ctrl) :: !weoi;
      toWakeUp := weoi :: !toWakeUp;
      sched ()

    let rml_get expr_evt f ctrl _ =
      rml_get_v (expr_evt()) f ctrl ()


(*************************************)
(* await                             *)
(*************************************)

    let rml_await_v evt k ctrl _ =
      rml_await_immediate_v evt (rml_pause k ctrl) ctrl ()

    let rml_await expr_evt k ctrl _ =
      let evt = expr_evt () in
      rml_await_v evt k ctrl ()


(*************************************)
(* await_all                         *)
(*************************************)

    let rml_await_all_v evt p ctrl _ =
      rml_await_immediate_v evt (rml_get_v evt p ctrl) ctrl ()

    let rml_await_all expr_evt p ctrl _ =
      let evt = expr_evt () in
      rml_await_all_v evt p ctrl ()
```



```


(*************************************)
(* await_all_match                   *)
(*************************************)

    let rml_await_all_match_v evt matching k ctrl _ =
      let rec self _ =
        rml_await_all_v evt
          (fun v () ->
             if matching v then k v ()
             else self ())
          ctrl ()
      in
      self ()

    let rml_await_all_match expr_evt matching k ctrl _ =
      let evt = expr_evt () in
      rml_await_all_match_v evt matching k ctrl ()


(*************************************)
(* await_immediate_one               *)
(*************************************)

    let rml_await_immediate_one_v evt f ctrl _ =
      rml_await_immediate_v evt
        (fun _ ->
           let (n, _, _) = evt in
           assert (Event.status n);
           let v = Event.one n in
           f v ())

    let rml_await_immediate_one expr_evt f ctrl _ =
      let evt = expr_evt() in
      rml_await_immediate_one_v evt f ctrl ()

(*************************************)
(* await_one                         *)
(*************************************)

    let rml_await_one expr_evt p ctrl _ =
      let pause_p x =
        rml_pause (fun () -> p x ()) ctrl
      in
      rml_await_immediate_one expr_evt pause_p ctrl ()

    let rml_await_one_v evt p ctrl _ =
      let pause_p x =
        rml_pause (fun () -> p x ()) ctrl
      in
      rml_await_immediate_one_v evt pause_p ctrl ()

    let rml_await_one_match expr_evt matching p =
      rml_await_all_match expr_evt
        (fun x -> List.exists matching x)
        (fun l ->
          try
            let v = List.find matching l in
            p v
          with Not_found -> raise RML)

    let rml_await_one_match_v evt matching p =
      rml_await_all_match_v evt
        (fun x -> List.exists matching x)
        (fun l ->
          try
            let v = List.find matching l in
            p v
```

```
          with Not_found -> raise RML)



(************************************)
(* signal                           *)
(************************************)

    let rml_signal p _ =
      let evt = new_evt() in
      let f = p evt in
      f ()

    let rml_signal_combine_v_v default comb p _ =
      let evt = new_evt_combine default comb in
      let f = p evt in
      f ()

    let rml_signal_combine_v_e default comb p _ =
      rml_signal_combine_v_v default (comb()) p ()

    let rml_signal_combine_e_v default comb p _ =
      rml_signal_combine_v_v (default()) comb p ()

    let rml_signal_combine default comb p _ =
      rml_signal_combine_v_v (default()) (comb()) p ()

    let rml_signal_memory_combine_v_v default comb p _ =
      let evt = new_evt_memory_combine default comb in
      let f = p evt in
      f ()

    let rml_signal_memory_combine_v_e default comb p _ =
      rml_signal_memory_combine_v_v default (comb()) p ()

    let rml_signal_memory_combine_e_v default comb p _ =
      rml_signal_memory_combine_v_v (default()) comb p ()

    let rml_signal_memory_combine default comb p _ =
      rml_signal_memory_combine_v_v (default()) (comb()) p ()


(************************************)
(* par                              *)
(************************************)
    let rml_split_par n f _ =
      let j = ref n in
      let k_list = f j in
      current := List.rev_append k_list !current;
      sched()

    let rml_join_par j k _ =
      decr j;
      if !j > 0 then
        sched ()
      else
        k ()

(************************************)
(* join_def                         *)
(************************************)
    let rml_join_def j v_ref get_values k v =
      decr j;
      v_ref := v;
      if !j > 0 then
        sched ()
      else
        k (get_values())
```



```
(************************************)
(* loop                             *)
(************************************)
    let rml_loop p =
      let f_1 = ref dummy_step in
      let f_loop = p (fun _ -> !f_1 ()) in
      f_1 := f_loop;
      f_loop

(************************************)
(* loop_n                           *)
(************************************)
    let rml_loop_n_v n p k =
      let cpt = ref 0 in
      let f_1 = ref dummy_step in
      let f_loop =
        p
          (fun _ ->
            if !cpt > 0 then
              (decr cpt; !f_1 ())
            else
              k ())
        in
        f_1 := f_loop;
        fun _ ->
          if n > 0 then
            (cpt := n - 1;
             f_loop ())
          else
            k ()

    let rml_loop_n e p k _ =
      let n = e() in
      rml_loop_n_v n p k ()

(************************************)
(* match                            *)
(************************************)

    let rml_match_v e f _ =
      f e ()

    let rml_match e f _ =
      let k = f (e()) in
      k ()

(************************************)
(* run                              *)
(************************************)

    let rml_run_v p k ctrl _ =
      p k ctrl ()

    let rml_run e k ctrl _ =
      (e ()) k ctrl ()

(************************************)
(* if                               *)
(************************************)

    let rml_if_v e k_1 k_2 _ =
      if e then
        k_1 ()
      else
        k_2 ()

    let rml_if e k_1 k_2 _ =
      rml_if_v (e()) k_1 k_2 ()
```

```
(*************************************)
(* while                             *)
(*************************************)

    let rml_while e p k =
      let f_body = ref dummy_step in
      let f_while _ =
        if e()
        then !f_body ()
        else k ()
      in
      f_body := p f_while;
      f_while

(*************************************)
(* for                               *)
(*************************************)

    let rml_for e1 e2 dir p =
      let (incr, cmp) = if dir then incr, (<=) else decr, (>=) in
      fun k ->
        let rec f_for i v2 =
          fun _ ->
            incr i;
            if cmp !i v2
            then p !i (f_for i v2) ()
            else k ()
        in
        let f_for_init =
          fun _ ->
            let i = ref (e1()) in
            let v2 = e2() in
            if cmp !i v2
            then p !i (f_for i v2) ()
            else k ()
        in
        f_for_init


(*************************************)
(* for_dopar                         *)
(*************************************)

    let rml_fordopar e1 e2 dir p k _ =
      if dir then
        begin
          let min = e1() in
          let max = e2() in
          let j = ref (max - min + 1) in
          if !j <= 0 then
            k ()
          else
            begin
              for i = max downto min do
                let f = p j i in
                current := f :: !current
              done;
              sched()
            end
        end
      else
        begin
          let max = e1() in
          let min = e2() in
          let j = ref (max - min + 1) in
          if !j <= 0 then
            k ()
```



```
          else
            begin
              for i = min to max do
                let f = p j i in
                current := f :: !current
              done;
              sched ()
            end
        end


(* ---------- Misc functions for until, control and when ---------- *)
    let new_ctrl kind cond =
      { kind = kind;
        alive = true;
        susp = false;
        children = [];
        cond = cond;
        next = [] }


(*************************************)
(* until                             *)
(*************************************)
    let rml_start_until_v ctrl (n,_,_) p k _ =
      let new_ctrl =
        new_ctrl
          (Kill (fun () -> let v = Event.value n in k v))
          (fun () -> Event.status n)
      in
      ctrl.children <- new_ctrl :: ctrl.children;
      p new_ctrl ()

    let rml_start_until ctrl expr_evt p k _ =
      rml_start_until_v ctrl (expr_evt ()) p k ()

    let rml_end_until new_ctrl k x =
      new_ctrl.alive <- false;
      k x

(*************************************)
(* control                           *)
(*************************************)
   let rml_start_control_v ctrl (n, _, _) p _ =
      let new_ctrl =
        new_ctrl
          Susp
          (fun () -> Event.status n)
      in
      ctrl.children <- new_ctrl :: ctrl.children;
      p new_ctrl ()

   let rml_start_control ctrl expr_evt p _ =
     rml_start_control_v ctrl (expr_evt()) p ()

    let rml_end_control new_ctrl k x =
      new_ctrl.alive <- false;
      k x


(*************************************)
(* when                              *)
(*************************************)
    let step_when ctrl new_ctrl n w =
      let rec f_when =
        fun _ ->
          if Event.status n
          then
            (new_ctrl.susp <- false;
```

```
             next_to_current new_ctrl;
             sched())
          else
            if !eoi
            then
              (ctrl.next <- f_when :: ctrl.next;
               sched())
            else
              (w := f_when :: !w;
               if ctrl.kind <> Top then toWakeUp := w :: !toWakeUp;
               sched())
      in f_when

    let rml_start_when_v ctrl (n,wa,wp) p _ =
      let dummy = ref (fun _ -> assert false) in
      let new_ctrl =
        new_ctrl
          (When dummy)
          (fun () -> Event.status n)
      in
      let _ = new_ctrl.susp <- true in
      let f_when =
        step_when ctrl new_ctrl n (if ctrl.kind = Top then wa else wp)
      in
      dummy := f_when;
      new_ctrl.next <- p new_ctrl :: new_ctrl.next;
      ctrl.children <- new_ctrl :: ctrl.children;
      f_when ()

    let rml_start_when ctrl expr_evt p _ =
      rml_start_when_v ctrl (expr_evt()) p ()

    let rml_end_when new_ctrl k x =
      new_ctrl.alive <- false;
      k x


(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
    exception End

(***********************************************)
(* rml_make                                    *)
(***********************************************)
    let rml_make p =
      let result = ref None in
      (* the main step function *)
      let f = p (fun x -> result := Some x; raise End) top in
      current := [f];
      (* the react function *)
      let rml_react () =
        try
          sched ();
          eoi := true;
          wakeUpAll ();
          sched ();
          eval_control_and_next_to_current ();
          Event.next ();
          eoi := false;
          None
        with
        | End -> !result
      in
      rml_react

  end
```