An Interpreter of DSL in ReactiveML and JoCaml

Louis Mandel
Université Paris-Sud 11
INRIA Parkas

1/12/2011 – Synchron 2011
Dynamic Synchronous Language (DSL)

Context

• ANR Partout
• language first proposed by Frédéric Boussinot and Jean-Ferdy Susini

DSL

• scripting language to the orchestration of concurrent tasks
• based on the reactive model of Boussinot and GALS
• multiple implementations
  – FunLoft, SugarCubes, ReactiveML/JoCaml, etc.
Idea of the implementation

Build an interpreter similar to an evaluator of arithmetical expression

```plaintext
type expr =
    | Const of int
    | Add of expr * expr
    | Sub of expr * expr
    | Mul of expr * expr
    | Div of expr * expr

let rec eval_expr e =
    match e with
    | Const n -> n
    | Add (e1, e2) -> eval_expr e1 + eval_expr e2
    | Sub (e1, e2) -> eval_expr e1 - eval_expr e2
    | Mul (e1, e2) -> eval_expr e1 * eval_expr e2
    | Div (e1, e2) -> eval_expr e1 / eval_expr e2
```
type script =
| S_nothing       (* do nothing *)
| S_print of string (* print a message *)
| S_cooperate    (* wait the next instant *)
| S_seq of script * script (* sequential composition *)
| S_par of script * script (* parallel composition *)
| S_if of expr * script * script (* conditional *)
| S_loop of script (* infinite loop *)
| S_repeat of expr * script (* finite loop *)
| S_generate of event_id (* signal emission *)
| S_await of event_id (* signal waiting *)
| S_watching of event_id * script (* preemption *)
| S_call of fun_id * const list (* call to an external function *)
| S_launch of module_id * const list (* call to an external process *)
| S_drop of site_id * script (* migration *)
let rec process eval_script script =
  match script with
  | S_nothing -> ()
  | S_print s -> print_endline s
  | S_cooperate -> pause
  | S_seq (s1, s2) ->
    run (eval_script s1);
    run (eval_script s2)
  | S_par (s1, s2) ->
    run (eval_script s1) ||
    run (eval_script s2)
...

Implementation in ReactiveML

let rec process eval_script script =
    match script with
    ...
    | S_generate ev_id ->
        let ev = event_of_event_id ev_id in
        emit ev
    ...

Allocation and dynamic binding of signals

let event_of_event_id =
    let tbl = Hashtbl.create 7 in
    fun ev_id ->
        try Hashtbl.find tbl ev_id
        with Not_found ->
            signal ev default () gather (fun () () -> ()) in
                Hashtbl.add tbl ev_id ev;
        ev
let rec process eval_script script =
  match script with
  ...
  | S_drop (site_id, script) ->
    Dsl_drop.put (site_id, script)
let put, get =
   def put(site_id_x_script) & state(to_drop) =
      reply () to put &
      state(site_id_x_script :: to_drop)
   or get() & state(to_drop) =
      reply to_drop to get &
      state([])
   in
   spawn state([]);
   put, get
let make_dsl_step main =
    let rml_react =
        Implem.Lco_ctrl_tree_record.rml_make_exec_process main
    in
    fun () ->
        let sl = get_to_add () in
        let v = rml_react (List.map Dsl_evaluator.eval_script sl) in
        exec_drop ();
    v
Conclusion

- Implementation of DSL for distributed architecture
- Interpreter and toplevel of DSL in less of 1500 SLOC
- Example of mixing JoCaml/ReactiveML and ReactiveML/JoCaml