rdbg: design choices

Erwan Jahier (Verimag, Grenoble, France)

October 29, 2019

Outline

1. rdbg : a programmable debugger for reactive programs
2. A motivating demo (Level 1)
3. Only a tiny API is necessary (Level 2)
4. Design choices for the kernel (Level 4)
5. Runtime Instrumentation (Level 3)
6. Conclusion

Plan

1. rdbg : a programmable debugger for reactive programs
2. A motivating demo (Level 1)
3. Only a tiny API is necessary (Level 2)
4. Design choices for the kernel (Level 4)
5. Runtime Instrumentation (Level 3)
6. Conclusion

`rdbg` is based on 3 orthogonal ideas

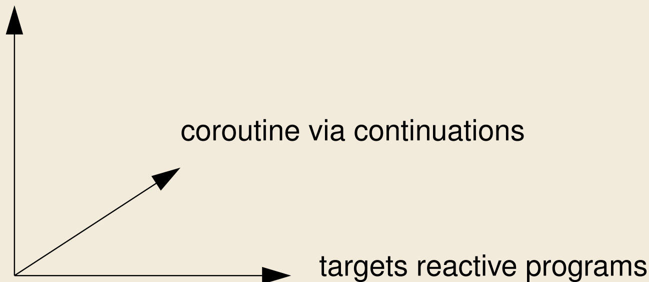
→ targets reactive programs

`rdbg` **is based on 3 orthogonal ideas**

Programmable

targets reactive programs

rdbg is based on 3 orthogonal ideas



Debugging Reactive Programs (dimension 1)

- Reactive systems – continuously interact with their environment
  - We need a lot of inputs
  - The feedback loop makes it difficult to provide realistic inputs offline

Debugging Reactive Programs (dimension 1)

- Reactive systems – continuously interact with their environment
  - We need a lot of inputs
  - The feedback loop makes it difficult to provide realistic inputs offline
  - Need to program reactive programs environments

Debugging Reactive Programs (dimension 1)

- Reactive systems – continuously interact with their environment
  - We need a lot of inputs
  - The feedback loop makes it difficult to provide realistic inputs offline
  - Need to program reactive programs environments
- Testing Reactive Programs
  - Lurette: the tool that performs all the red tape (plumbering) between the SUT, its env, and the oracles
  - Lutin: a language to program non-deterministic reactive systems

Debugging Reactive Programs (dimension 1)

- Reactive systems – continuously interact with their environment
  - We need a lot of inputs
  - The feedback loop makes it difficult to provide realistic inputs offline
  - Need to program reactive programs environments
- Testing Reactive Programs
  - Lurette: the tool that performs all the red tape (plumbering) between the SUT, its env, and the oracles
  - Lutin: a language to program non-deterministic reactive systems
- The idea was to reuse the Lurette infrastructure

Debugging Reactive Programs (dimension 1)

- Reactive systems – continuously interact with their environment
  - We need a lot of inputs
  - The feedback loop makes it difficult to provide realistic inputs offline
  - Need to program reactive programs environments
- Testing Reactive Programs
  - Lurette: the tool that performs all the red tape (plumbering) between the SUT, its env, and the oracles
  - Lutin: a language to program non-deterministic reactive systems
- The idea was to reuse the Lurette infrastructure
  - to provide inputs

Debugging Reactive Programs (dimension 1)

- Reactive systems – continuously interact with their environment
  - We need a lot of inputs
  - The feedback loop makes it difficult to provide realistic inputs offline
  - Need to program reactive programs environments
- Testing Reactive Programs
  - Lurette: the tool that performs all the red tape (plumbering) between the SUT, its env, and the oracles
  - Lutin: a language to program non-deterministic reactive systems
- The idea was to reuse the Lurette infrastructure
  - to provide inputs
  - When a bug is found during testing, everything is ready for debug

Debugging Reactive Programs (dimension 1)

- Reactive systems – continuously interact with their environment
  - We need a lot of inputs
  - The feedback loop makes it difficult to provide realistic inputs offline
  - Need to program reactive programs environments
- Testing Reactive Programs
  - Lurette: the tool that performs all the red tape (plumbering) between the SUT, its env, and the oracles
  - Lutin: a language to program non-deterministic reactive systems
- The idea was to reuse the Lurette infrastructure
  - to provide inputs
  - When a bug is found during testing, everything is ready for debug
  - Oracles sometimes need to be debugged

rdbg: a Programmable debugger (dimension 2)

- Why programmable?
  - because debugger users are programmers (always)
  - “tailorisation”

`rdbg`: a Programmable debugger (dimension 2)

- Why programmable?
  - because debugger users are programmers (always)
  - “tailorisation”
  - Moreover it’s not hard to do!

rdbg: a Programmable debugger (dimension 2)

- Why programmable?
  - because debugger users are programmers (always)
  - “tailorisation”
  - Moreover it’s not hard to do!

Claim: Given

1. a host language (with good libs and REPL)

`rdbg`: a Programmable debugger (dimension 2)

- Why programmable?
  - because debugger users are programmers (always)
  - “tailorisation”
  - Moreover it’s not hard to do!

Claim: Given

1. a host language (with good libs and REPL)
2. a pre-defined instrumentation of the debuggee process,

`rdbg`: a Programmable debugger (dimension 2)

- Why programmable?
  - because debugger users are programmers (always)
  - “tailorisation”
  - Moreover it’s not hard to do!

Claim: Given

1. a host language (with good libs and REPL)
2. a pre-defined instrumentation of the debuggee process,

it’s easy to build a extensible debugger on top of a tiny API

`rdbg`: a Programmable debugger (dimension 2)

- Why programmable?
  - because debugger users are programmers (always)
  - “tailorisation”
  - Moreover it’s not hard to do!

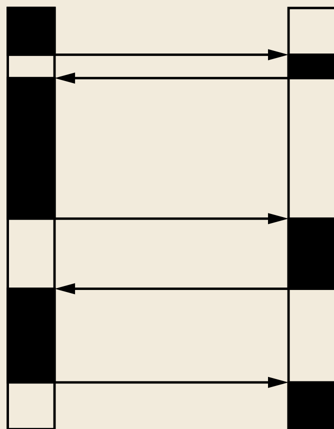
Claim: Given

1. a host language (with good libs and REPL)
2. a pre-defined instrumentation of the debuggee process,

it’s easy to build a extensible debugger on top of a tiny API $\rightarrow$ separation of concerns

Coroutine via Continuations (dimension 3)

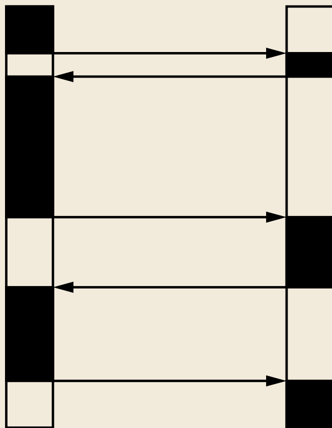
- Coroutine between
  Debugger - debuggee



- Usually implemented via 2 processes communicating via
  - signals/ptrace (gdb)
  - pipes or sockets (camldebug, jdwp, ...)
- pb : context switches cost

Coroutine via Continuations (dimension 3)

- Coroutine between Debugger - debuggee



- Usually implemented via 2 processes communicating via
  - signals/ptrace (gdb)
  - pipes or sockets (camldebug, jdwp, ...)
- pb : context switches cost
- using continuations : only one process

4 levels of Stakeholders

- Level 1: End-users can use existing commands

4 levels of Stakeholders

- Level 1: End-users can use existing commands
- Level 2: Advanced users can program debugging commands

4 levels of Stakeholders

- Level 1: End-users can use existing commands
- Level 2: Advanced users can program debugging commands
- Level 3: Compiler experts provide the runtime instrumentation (a/k.a., rdbg plugins)

4 levels of Stakeholders

- Level 1: End-users can use existing commands
- Level 2: Advanced users can program debugging commands
- Level 3: Compiler experts provide the runtime instrumentation (a/k.a., rdbg plugins)
- Level 4: The framework designer (bb)

Plan

1. rdbg : a programmable debugger for reactive programs
2. A motivating demo (Level 1)
3. Only a tiny API is necessary (Level 2)
4. Design choices for the kernel (Level 4)
5. Runtime Instrumentation (Level 3)
6. Conclusion

Plan

1 rdbg : a programmable debugger for reactive programs

2 A motivating demo (Level 1)

3 Only a tiny API is necessary (Level 2)

4 Design choices for the kernel (Level 4)

5 Runtime Instrumentation (Level 3)

6 Conclusion

demo

```
cd demo/heater
make clean ; ls
cat Makefile
make sut
make env
make lurette
make rdbg
n
ni 10
b
bi 6
s si
g 23
cg
cgf
cd demo/lutin-tuto
n n n n s
```

Plan

1. rdbg : a programmable debugger for reactive programs
2. A motivating demo (Level 1)
3. Only a tiny API is necessary (Level 2)
4. Design choices for the kernel (Level 4)
5. Runtime Instrumentation (Level 3)
6. Conclusion

The `rdbg` API

The whole user API lies in the definition of `rdbg` event (`Event.t`)

The `rdbg` API

The whole user API lies in the definition of `rdbg` event (`Event.t`)

- Source level info
  - node/function name
  - language (e.g., “lutin” or “lustre”)
  - input variables (names and types)
  - input variables
  - lazily: file names, line numbers, call stack

The `rdbg` API

The whole user API lies in the definition of `rdbg` event (`Event.t`)

- Source level info
  - node/function name
  - language (e.g., “lutin” or “lustre”)
  - input variables (names and types)
  - input variables
  - lazily: file names, line numbers, call stack
- Runtime info
  - Event number
  - Step number;
  - Execution depth
  - Data state
  - Event kind (Call, Exit, Internal)

The `rdbg` API

The whole user API lies in the definition of `rdbg` event (`Event.t`)

- Source level info
  - node/function name
  - language (e.g., “lutin” or “lustre”)
  - input variables (names and types)
  - input variables
  - lazily: file names, line numbers, call stack
- Runtime info
  - Event number
  - Step number;
  - Execution depth
  - Data state
  - Event kind (Call, Exit, Internal)
- Navigation
  - Go to the next event ;

The `rdbg` API

The whole user API lies in the definition of `rdbg` event (`Event.t`)

- Source level info
  - node/function name
  - language (e.g., “lutin” or “lustre”)
  - input variables (names and types)
  - input variables
  - lazily: file names, line numbers, call stack
- Runtime info
  - Event number
  - Step number;
  - Execution depth
  - Data state
  - Event kind (Call, Exit, Internal)
- Navigation
  - Go to the next event ;
  - Terminate

The rdbg API

The whole user API lies in the definition of rdbg event (Event.t)

- Source level info
  - node/function name
  - language (e.g., “lutin” or “lustre”)
  - input variables (names and types)
  - input variables
  - lazily: file names, line numbers, call stack
- Runtime info
  - Event number
  - Step number;
  - Execution depth
  - Data state
  - Event kind (Call, Exit, Internal)
- Navigation
  - Go to the next event ;
  - Terminate

```
run :  session_args -> Event.t
```

The `rdbg` API

The whole user API lies in the definition of `rdbg` event (`Event.t`)

- Source level info
  - node/function name
  - language (e.g., “lutin” or “lustre”)
  - input variables (names and types)
  - input variables
  - lazily: file names, line numbers, call stack
- Runtime info
  - Event number
  - Step number;
  - Execution depth
  - Data state
  - Event kind (Call, Exit, Internal)
- Navigation
  - Go to the next event ;
  - Terminate

```
run :  session_args -> Event.t
```

and that’s all!

The `rdbg ocaml` API

```
type kind = Ltop | Call | Exit | MicroStep of string

type t = {
  name : string; (* node name *)
  lang : string; (* e.g., "lutin" or "lustre" *)
  inputs  : var list;
  outputs : var list;
  sinfo : (unit -> src_info) option;
(* Runtime info *)
  nb     : int;
  step   : int;
  depth  : int;
  data   : Data.subst list;
  kind   : kind; (* Call, Exit, MicroStep *)
(* Navigation *)
  next   : unit -> t;
  terminate: unit -> unit }
```

`run :  args -> Event.t`

that's all!

What can be done with this small kernel

- test/debug/monitor within the same framework
- Move forward and Backwards
- Conditional breakpoints
- `gdb` like Breakpoints
- Debugger Customisation
  - Adding hooks
  - custom traces
  - custom commands
- Profiling, monitoring
- Opening an emacs at the current line
- Computing CFG
- Experimenting coverage criterion
- ...

Step forward

```
let (next:Event.t -> Event.t)= fun e -> e.next ()
```

Step forward

```
let (next:Event.t -> Event.t)= fun e -> e.next ()
```

```
let (next:Event.t -> Event.t)= fun e -> print_event e; e.next ()
```

Step forward

```
let (next:Event.t -> Event.t)= fun e -> e.next ()
```

```
let (next:Event.t -> Event.t)= fun e -> print_event e; e.next ()
```

```
let rec (nexti:Event.t -> int -> Event.t)= fun e cpt ->
    if cpt > 0 then nexti (next e) (cpt-1) else e
```

Step forward

```
let (next:Event.t -> Event.t)= fun e -> e.next ()
```

```
let (next:Event.t -> Event.t)= fun e -> print_event e; e.next ()
```

```
let rec (nexti:Event.t -> int -> Event.t)= fun e cpt ->
    if cpt > 0 then nexti (next e) (cpt-1) else e
```

```
let rec (goto_i:Event.t -> int -> Event.t)= fun e i ->
    if e.nb < i then goto_i (next e) i else e
```

Step forward

```
let (next:Event.t -> Event.t)= fun e -> e.next ()
```

```
let (next:Event.t -> Event.t)= fun e -> print_event e; e.next ()
```

```
let rec (nexti:Event.t -> int -> Event.t)= fun e cpt ->
    if cpt > 0 then nexti (next e) (cpt-1) else e
```

```
let rec (goto_i:Event.t -> int -> Event.t)= fun e i ->
    if e.nb < i then goto_i (next e) i else e
```

```
let rec (goto_s:Event.t -> int -> Event.t)= fun e i ->
    if e.step < i then goto_s (next e) i else e
```

Demo

Conditional breakpoints

```
let rec (next_cond:Event.t -> (Event.t -> bool) -> Event.t) =
  fun e cond ->
```

Conditional breakpoints

```
let rec (next_cond:Event.t -> (Event.t -> bool) -> Event.t) =
  fun e cond ->
    let ne = next e in
```

Conditional breakpoints

```
let rec (next_cond:Event.t -> (Event.t -> bool) -> Event.t) =
  fun e cond ->
    let ne = next e in
      if cond ne then ne
```

Conditional breakpoints

```
let rec (next_cond:Event.t -> (Event.t -> bool) -> Event.t) =
  fun e cond ->
    let ne = next e in
      if cond ne then ne
      else next_cond ne cond
```

ex :

- spot the time when an invariant is violated

Conditional breakpoints

```
let rec (next_cond:Event.t -> (Event.t -> bool) -> Event.t) =
  fun e cond ->
    let ne = next e in
      if cond ne then ne
      else next_cond ne cond
```

ex :

- spot the time when an invariant is violated
- implement the step (s) command seen during the motivating demo

```
let s e = next_cond e (fun ne -> ne.kind=Exit && ne.name = e.name
                                             && ne.depth = e.depth)
```

Demo

Moving backwards

```
let rev_cond e (p:Event.t->bool) =
```

Moving backwards

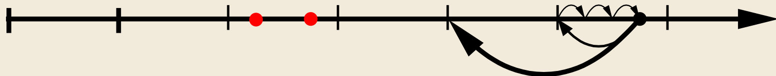
```
let rev_cond e (p:Event.t->bool) =
```

Moving backwards

```
let rev_cond e (p:Event.t->bool) =
```

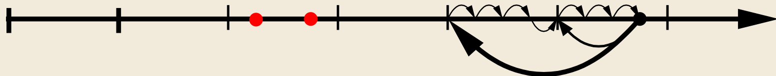
Moving backwards

```
let rev_cond e (p:Event.t->bool) =
```



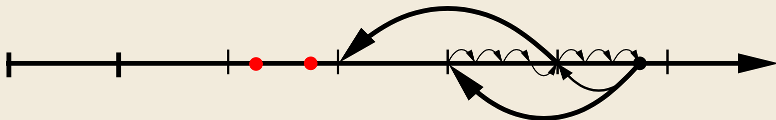
Moving backwards

```
let rev_cond e (p:Event.t->bool) =
```



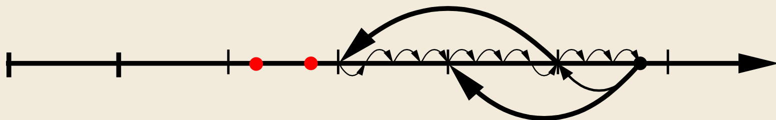
Moving backwards

```
let rev_cond e (p:Event.t->bool) =
```



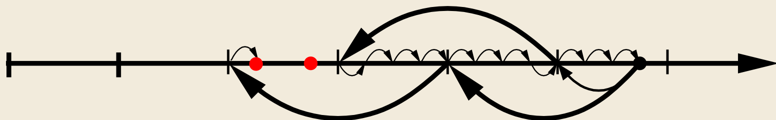
Moving backwards

```
let rev_cond e (p:Event.t->bool) =
```



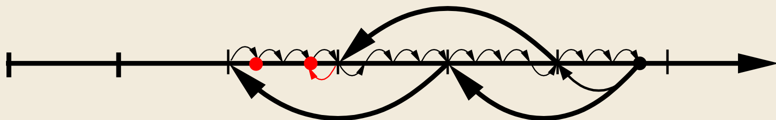
Moving backwards

```
let rev_cond e (p:Event.t->bool) =
```



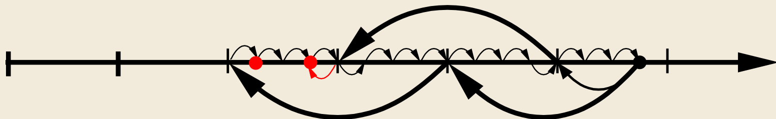
Moving backwards

```
let rev_cond e (p:Event.t->bool) =
```



Moving backwards

```
let rev_cond e (p:Event.t->bool) =
```



```
let (rev_cond : Event.t -> (Event.t -> bool) -> Event.t) = fun e p ->
  let rec aux2 e i = (* search if there exist an event e2 in [e.nb;i[ s.t. p e2
      otherwise, recursive search into [e.nb-check_rate; e.nb[  *)
    assert (e.nb<i);
    let e = if p e then e else next_cond e (fun e -> p e || e.nb = i) in
      if p e then aux3 e e i (* search for a more recent event satisfying p *) else (* e.nb=i *)
        let x = (e.nb / !ckpt_rate) -1 in
        if x < 0 then find_ckpt 0 else aux2 (find_ckpt x) ((x+1) * !ckpt_rate -1)
  and aux3 e e_good i = (* search if there exist an event e in ]e_good.n;i[ s.t. p e *)
    if e.nb = i then e_good else
        let e = next_cond e (fun e -> p e || e.nb = i) in
        let e_good = if p e then e else e_good in
          aux3 e e_good i
  in
    if e.nb = 1 then failwith "Cannot move backwards from the first event.\n" else
      let x = ((e.nb - 1)/ !ckpt_rate) in
      let last_e = find_ckpt x in
      if last_e.nb = e.nb-1 then aux2 (find_ckpt (x-1)) (e.nb-1) else aux2 last_e (e.nb-1)
(* move backwards until a breakpoint is reached *)
let (rev : Event.t -> Event.t) = fun e ->
  let stop e = List.exists (break_matches e) !breakpoints in rev_cond e stop
```

Adding hooks to `next`

```
let (hooks: (string * (Event.t -> unit)) list ref) =
  ref [("print_event",print_event)]
```

Adding hooks to `next`

```
let (hooks: (string * (Event.t -> unit)) list ref) =
  ref [("print_event",print_event)]

let rec (next : Event.t -> Event.t) =
  fun e ->
    let ne = e.next () in
       List.iter (fun (_,f) -> f ne) !hooks;
       ne
```

Adding hooks to `next`

```
let (hooks: (string * (Event.t -> unit)) list ref) =
  ref [("print_event",print_event)]

let rec (next : Event.t -> Event.t) =
  fun e ->
    let ne = e.next () in
       List.iter (fun (_,f) -> f ne) !hooks;
       ne
```

```
let (add_hooks: string * (Event.t -> unit) -> unit) =
  fun h -> hooks := (h::!hooks)

let (del_hooks: string -> unit) =
  fun str -> hooks := List.remove_assoc str !hooks
```

Custom traces

```
let my_print_event e =
  [...] ;;

del_hooks "print_event";;
add_hooks ("print_event",my_print_event);;
```

Demo

Shortcuts

```
let e = ref (run())
let n () = e:=next !e
let ni i = e:=nexti !e i
let r () = e:=ref (run())
...
```

Monitoring

```
let cpt = ref 0
let rec count e =
  match getb_val "b", getr_val "x" in
    | Some b, Some x -> if (b and 3 < x and x < 10)
                        then c:=c+1;
    | _,_ -> ();
  count (e.next())
```

Everything can also be compiled to native code for efficiency

gdb like Breakpoints

Set breakpoints via strings the form: “node” or “file::line”

```
let brkpts = ref []
```

gdb like Breakpoints

Set breakpoints via strings the form: “node” or “file::line”

```
let brkpts = ref []

let (break : string -> unit) = fun str ->
```

gdb like Breakpoints

Set breakpoints via strings the form: “node” or “file::line”

```
let brkpts = ref []

let (break : string -> unit) = fun str ->
  brkpts := str::!brkpts
```

gdb like Breakpoints

Set breakpoints via strings the form: “node” or “file::line”

```
let brkpts = ref []

let (break : string -> unit) = fun str ->
  brkpts := str::!brkpts

let (delete : unit -> unit) = fun () -> brkpts := []
```

gdb like Breakpoints

Set breakpoints via strings the form: “node” or “file::line”

```
let brkpts = ref []

let (break : string -> unit) = fun str ->
  brkpts := str::!brkpts

let (delete : unit -> unit) = fun () -> brkpts := []

let (continue : Event.t -> Event.t) =
```

gdb like Breakpoints

Set breakpoints via strings the form: “node” or “file::line”

```
let brkpts = ref []

let (break : string -> unit) = fun str ->
  brkpts := str::!brkpts

let (delete : unit -> unit) = fun () -> brkpts := []

let (continue : Event.t -> Event.t) =  fun e ->
    let stop e = List.exists (brk_matches e) !brkpts in
```

gdb like Breakpoints

Set breakpoints via strings the form: “node” or “file::line”

```
let brkpts = ref []

let (break : string -> unit) = fun str ->
  brkpts := str::!brkpts

let (delete : unit -> unit) = fun () -> brkpts := []

let (continue : Event.t -> Event.t) =  fun e ->
    let stop e = List.exists (brk_matches e) !brkpts in
      next_cond e stop
```

gdb like Breakpoints (cont)

```
let (brk_matches : Event.t -> string -> bool) =
fun e b ->
```

gdb like Breakpoints (cont)

```
let (brk_matches : Event.t -> string -> bool) =
fun e b ->
 let si_match_file str si =
   (si.file = str || basename si.file = str) in
```

gdb like Breakpoints (cont)

```
let (brk_matches : Event.t -> string -> bool) =
fun e b ->
 let si_match_file str si =
   (si.file = str || basename si.file = str) in
 let si_match_line ln {line=(d,f)} =
  (d <= ios && ios i <= f) in
```

gdb like Breakpoints (cont)

```
let (brk_matches : Event.t -> string -> bool) =
fun e b ->
 let si_match_file str si =
   (si.file = str || basename si.file = str) in
 let si_match_line ln {line=(d,f)} =
  (d <= ios && ios i <= f) in
  match e.sinfo, Str.split (Str.regexp "::") b with
```

gdb like Breakpoints (cont)

```
let (brk_matches : Event.t -> string -> bool) =
fun e b ->
 let si_match_file str si =
   (si.file = str || basename si.file = str) in
 let si_match_line ln {line=(d,f)} =
  (d <= ios && ios i <= f) in
  match e.sinfo, Str.split (Str.regexp "::") b with
  | None, _ |  _, [] -> false (* no more BP *)
```

gdb like Breakpoints (cont)

```
let (brk_matches : Event.t -> string -> bool) =
fun e b ->
 let si_match_file str si =
   (si.file = str || basename si.file = str) in
 let si_match_line ln {line=(d,f)} =
  (d <= ios && ios i <= f) in
  match e.sinfo, Str.split (Str.regexp "::") b with
  | None, _ |  _, [] -> false (* no more BP *)
  | Some src, str::tail ->  let src = src() in
```

gdb like Breakpoints (cont)

```
let (brk_matches : Event.t -> string -> bool) =
fun e b ->
 let si_match_file str si =
   (si.file = str || basename si.file = str) in
 let si_match_line ln {line=(d,f)} =
  (d <= ios && ios i <= f) in
  match e.sinfo, Str.split (Str.regexp "::") b with
  | None, _ |  _, [] -> false (* no more BP *)
  | Some src, str::tail ->  let src = src() in
    match tail with
     | [ln] -> List.exists
       (fun si -> si_match_file str si &&
                  si_match_line ln si) src.atoms
     | [] -> List.exists (si_match_file str) src.atoms
     | _  -> false
```

Demo

Profiling

```
let prof_tbl = Hashtbl.create 50
```

Profiling

```
let prof_tbl = Hashtbl.create 50
let incr_prof si =
  try let cpt = Hashtbl.find prof_tbl si in
      Hashtbl.replace prof_tbl si (cpt+1)
```

Profiling

```
let prof_tbl = Hashtbl.create 50
let incr_prof si =
  try let cpt = Hashtbl.find prof_tbl si in
      Hashtbl.replace prof_tbl si (cpt+1)
  with Not_found -> Hashtbl.add prof_tbl si 1
```

Profiling

```
let prof_tbl = Hashtbl.create 50
let incr_prof si =
  try let cpt = Hashtbl.find prof_tbl si in
      Hashtbl.replace prof_tbl si (cpt+1)
  with Not_found -> Hashtbl.add prof_tbl si 1

let (prof_add: Event.t -> unit) =
  fun e ->
    match to_lut_evt e.kind, e.sinfo with
      | (Sat | Nsat), Some src ->
          List.iter incr_prof (src()).atoms
      | _ -> ()
let set_profiler on =
  if on then add_hooks ("profile",prof_add)
        else del_hooks "profile"
```

Demo

Time Profiling

- Save the `Unix.time()` at `call` events
- Accumulate the difference at `exit` events

Computing CFG

cf file:demo/demo.org

Experimenting with Coverage criterion

cf file:demo/demo.org

Plan

1. rdbg : a programmable debugger for reactive programs
2. A motivating demo (Level 1)
3. Only a tiny API is necessary (Level 2)
4. **Design choices for the kernel (Level 4)**
5. Runtime Instrumentation (Level 3)
6. Conclusion

Choice of the host language (1)

- We need
  - real/eval/print/loop toplevel interpreter
  - good support (librairies, tools, community)

Choice of the host language (1)

- We need
  - real/eval/print/loop toplevel interpreter
  - good support (librairies, tools, community)
- There is no perfect choice: `Ocaml` is not that bad
- Only users who want to add debugging commands need to know `Ocaml`

Defining Reactive Events (2)

- An event is an observation point Closely related to the language semantics

Defining Reactive Events (2)

- An event is an observation point Closely related to the language semantics
- Defining event should rather be a concern of the language plugin developper, but
- We want events that are versatile enough to capture any reactive languages operational semantics!

Defining Reactive Events (2)

- An event is an observation point Closely related to the language semantics
- Defining event should rather be a concern of the language plugin developper, but
- We want events that are versatile enough to capture any reactive languages operational semantics!

Well, at least on any synchronous languages operational semantics

…

Defining Reactive Events (2)

- An event is an observation point Closely related to the language semantics
- Defining event should rather be a concern of the language plugin developper, but
- We want events that are versatile enough to capture any reactive languages operational semantics!

Well, at least on any synchronous languages operational semantics
. . . Well, at least on Lustre and Lutin!

Defining Reactive Events (2)

- An event is an observation point Closely related to the language semantics
- Defining event should rather be a concern of the language plugin developper, but
- We want events that are versatile enough to capture any reactive languages operational semantics!

Well, at least on any synchronous languages operational semantics
... Well, at least on Lustre and Lutin!

- `Call` : when entering a node
- `Exit` : when exiting a node
- `Microstep` : holds info related to the lang micro-step

Defining Reactive Events (2)

- An event is an observation point Closely related to the language semantics
- Defining event should rather be a concern of the language plugin developper, but
- We want events that are versatile enough to capture any reactive languages operational semantics!

Well, at least on any synchronous languages operational semantics
. . . Well, at least on Lustre and Lutin!

- `Call` : when entering a node
- `Exit` : when exiting a node
- `Microstep` : holds info related to the lang micro-step
- Events specific to Lutin:
  - `microstep("try")` / `microstep("sat")` / `microstep("unsat")` : a choice is performed in the program control structure

Debugger/debuggee communications (3)

- Signals+Runtime instrumentation (gdb, ocamldebug)
  - fixed set of commands
- Pipes, sockets (javadbg)
  - watch out the context switches
- Continuations
  - No msg writing/parsing between the 2 entities
  - No context switches, which matters for debugging programs that interact at each step with the debuggee
  - Very easy instrumentation (once the code is CPS...)

Coroutine via CPS: From Lurette to rdbg [1/4]

```
-- Vanilla lurette top loop
let p a b = a+b
let step_env a = p a 1
let step_sut a = p a 2
let rec loop i a =
 if i>42 then
  raise (End a) else
  let b = step_env a in
  let c = step_sut b in
   loop (i+1) c
```

Coroutine via CPS: From Lurette to rdbg [2/4]

```
-- lurettop loop              -- CPS version
let p a b = a+b               let p a b ct = ct (a+b)
```

Coroutine via CPS: From Lurette to rdbg [2/4]

```
-- lurettop loop            -- CPS version
let p a b = a+b             let p a b ct = ct (a+b)
let step_env a = p a 1      let step_env a ct = p a 1 ct
```

Coroutine via CPS: From Lurette to rdbg [2/4]

```
-- lurettop loop            -- CPS version
let p a b = a+b             let p a b ct = ct (a+b)
let step_env a = p a 1      let step_env a ct = p a 1 ct
let step_sut a = p a 2      let step_sut a ct = p a 2 ct
```

Coroutine via CPS: From Lurette to rdbg [2/4]

```
-- lurettop loop              -- CPS version
let p a b = a+b               let p a b ct = ct (a+b)
let step_env a = p a 1        let step_env a ct = p a 1 ct
let step_sut a = p a 2        let step_sut a ct = p a 2 ct
let rec loop i a =            let rec loop i a =
 if i>42 then                  if i > 42 then
  raise (End a) else            raise (End a) else
  let b = step_env a in         step_env a (loop2 i)
```

Coroutine via CPS: From Lurette to rdbg [2/4]

```
-- lurettop loop                -- CPS version
let p a b = a+b                 let p a b ct = ct (a+b)
let step_env a = p a 1          let step_env a ct = p a 1 ct
let step_sut a = p a 2          let step_sut a ct = p a 2 ct
let rec loop i a =              let rec loop i a =
 if i>42 then                    if i > 42 then
  raise (End a) else              raise (End a) else
  let b = step_env a in           step_env a (loop2 i)
  let c = step_sut b in         and loop2 i b = step_sut b
   loop (i+1) c                                  (loop (i+1))
```

Coroutine via CPS: From Lurette to rdbg [3/4]

The CPS functions never return
Hence it's easy to add events at each event locations

```
type event =
  {
    next : unit -> event;
    msg  : string ;
 -- step:int;
 -- depth:int;
 -- data: substs;
 -- src: string;
 -- ...
}
```

Coroutine via CPS: From Lurette to rdbg [4/4]

```
let plus a b cont =

                   cont (a+b)
let step_env a cont =

                   plus a 1 cont
let step_sut a cont =

                   plus a 2 cont
let rec loop2 i b = step_sut b (loop (i+1))
and loop i a =
  if a > 42 then raise (End a)
            else
                  step_env a (loop2 i)
```

Coroutine via CPS: From Lurette to rdbg [4/4]

```
let plus a b cont =


                   cont (a+b)
let step_env a cont = {
  msg  = "step_env" ;
  next = fun () -> plus a 1 cont }
let step_sut a cont =


                   plus a 2 cont
let rec loop2 i b = step_sut b (loop (i+1))
and loop i a =
  if a > 42 then raise (End a)
            else
                  step_env a (loop2 i)
```

Coroutine via CPS: From Lurette to rdbg [4/4]

```
let plus a b cont =


                   cont (a+b)
let step_env a cont = {
  msg  = "step_env" ;
  next = fun () -> plus a 1 cont }
let step_sut a cont = {
  msg  = "step_sut" ;
  next = fun () -> plus a 2 cont }
let rec loop2 i b = step_sut b (loop (i+1))
and loop i a =
  if a > 42 then raise (End a)
            else
                  step_env a (loop2 i)
```

Coroutine via CPS: From Lurette to rdbg [4/4]

```
let plus a b cont =  {
  msg  = sprintf("%i+%i" a b);
  next = fun () -> cont (a+b) }
let step_env a cont = {
  msg  = "step_env" ;
  next = fun () -> plus a 1 cont }
let step_sut a cont = {
  msg  = "step_sut" ;
  next = fun () -> plus a 2 cont }
let rec loop2 i b = step_sut b (loop (i+1))
and loop i a =
  if a > 42 then raise (End a)
          else
                step_env a (loop2 i)
```

Coroutine via CPS: From Lurette to rdbg [4/4]

```
let plus a b cont =  {
  msg  = sprintf("%i+%i" a b);
  next = fun () -> cont (a+b) }
let step_env a cont = {
  msg  = "step_env" ;
  next = fun () -> plus a 1 cont }
let step_sut a cont = {
  msg  = "step_sut" ;
  next = fun () -> plus a 2 cont }
let rec loop2 i b = step_sut b (loop (i+1))
and loop i a =
  if a > 42 then raise (End a)
            else  { msg  = "top";
  next = fun () ->step_env a (loop2 i)}
```

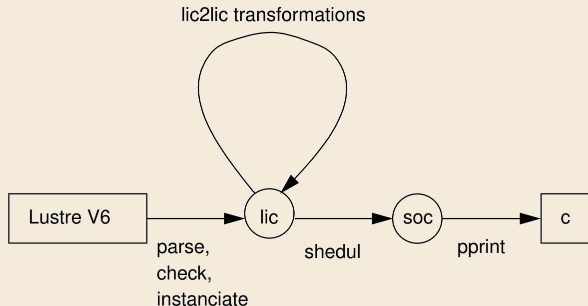
Plan

1. rdbg : a programmable debugger for reactive programs
2. A motivating demo (Level 1)
3. Only a tiny API is necessary (Level 2)
4. Design choices for the kernel (Level 4)
5. **Runtime Instrumentation (Level 3)**
6. Conclusion

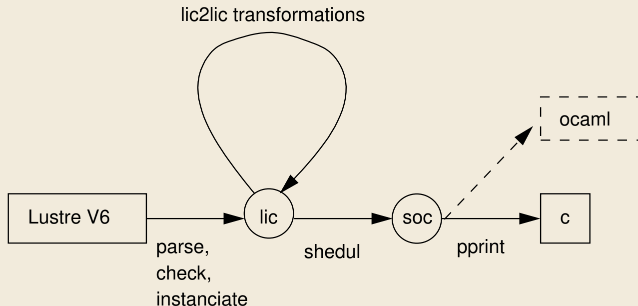
Design choices for the runtime instrumentation (Level 3)

- What to instrument?
  - an interpreter
  - the source/binary/intermediate code
  - runtime or static (by the compiler) instrumentation
- Granularity of the instrumentation: trade-off between efficiency and completeness

The V6 compilation process



The V6 compilation process



The rdbg Language Plugin Mechanism

To plug onto rdbg, one just need to provide cma/cmxs (dynamic ocamlopt code library) that implements this interface:

```
type sl = (string * Data.v) list (* substitutions *)
type e = Event.t (* a shortcut for the remaining *)
```

The `rdbg` Language Plugin Mechanism

To plug onto `rdbg`, one just need to provide cma/cmxs (dynamic ocamlopt code library) that implements this interface:

```
type sl = (string * Data.v) list (* substitutions *)
type e = Event.t (* a shortcut for the remaining *)
type t = {
 inputs:(Data.ident*Data.t) list; (* name and type *)
 outputs : (Data.ident * Data.t) list; (* ditto *)
 init_inputs  : sl;
 init_outputs : sl;
```

The rdbg Language Plugin Mechanism

To plug onto `rdbg`, one just need to provide cma/cmxs (dynamic ocamlopt code library) that implements this interface:

```
type sl = (string * Data.v) list (* substitutions *)
type e = Event.t (* a shortcut for the remaining *)
type t = {
 inputs:(Data.ident*Data.t) list; (* name and type *)
 outputs : (Data.ident * Data.t) list; (* ditto *)
 init_inputs  : sl;
 init_outputs : sl;
 step     : (sl -> sl); (* Lurette step *)
```

The rdbg Language Plugin Mechanism

To plug onto `rdbg`, one just need to provide cma/cmxs (dynamic ocamlopt code library) that implements this interface:

```
type sl = (string * Data.v) list (* substitutions *)
type e = Event.t (* a shortcut for the remaining *)
type t = {
 inputs:(Data.ident*Data.t) list; (* name and type *)
 outputs : (Data.ident * Data.t) list; (* ditto *)
 init_inputs  : sl;
 init_outputs : sl;
 step     : (sl -> sl); (* Lurette step *)
 step_dbg : (sl -> e -> ( sl -> e -> e) -> e); (* RDBG step *)
```

The rdbg Language Plugin Mechanism

To plug onto rdbg, one just need to provide cma/cmxs (dynamic ocamlopt code library) that implements this interface:

```
type sl = (string * Data.v) list (* substitutions *)
type e = Event.t (* a shortcut for the remaining *)
type t = {
 inputs:(Data.ident*Data.t) list; (* name and type *)
 outputs : (Data.ident * Data.t) list; (* ditto *)
 init_inputs  : sl;
 init_outputs : sl;
 step     : (sl -> sl); (* Lurette step *)
 step_dbg : (sl -> e -> ( sl -> e -> e) -> e); (* RDBG step *)
 kill: string -> unit;
}
```

Instrumenting soc

```
soc foo(i1, ..., im) returns (o1,..., on);
let
  ... -- some equations
tel
soc bar(...) returns(...);
let
  ...;
  (x1,...,xn) = foo(y1,...,ym); -- how to evaluate this expression?
  ...;
tel
```

Instrumenting soc

```
soc foo(i1, ..., im) returns (o1,..., on);
let
  ... -- some equations
tel
soc bar(...) returns(...);
let
  ...;
  (x1,...,xn) = foo(y1,...,ym); -- how to evaluate this expression?
  ...;
tel
```

- node `bar` is compiled into a `soc`, and:
  - `soc.ins` = `(i1, ..., im)` ; `soc.out` = `(o1,..., on)`
  - `argsO` = `(x1,...,xn)` ; `argsI` = `(y1,...,ym)`

Instrumenting soc

```
soc foo(i1, ..., im) returns (o1,..., on);
let
  ... -- some equations
tel
soc bar(...) returns(...);
let
  ...;
  (x1,...,xn) = foo(y1,...,ym); -- how to evaluate this expression?
  ...;
tel
```

- node bar is compiled into a soc, and:
  - soc.ins = (i1, ..., im) ; soc.out = (o1,..., on)
  - argsO = (x1,...,xn) ; argsI = (y1,...,ym)

```
let soc_step soc argsI argsO env = -- where "env" holds variables
    values
```

Instrumenting soc

```
soc foo(i1, ..., im) returns (o1,..., on);
let
  ... -- some equations
tel
soc bar(...) returns(...);
let
  ...;
  (x1,...,xn) = foo(y1,...,ym); -- how to evaluate this expression?
  ...;
tel
```

- node bar is compiled into a soc, and:
  - soc.ins = (i1, ..., im) ; soc.out = (o1,..., on)
  - args0 = (x1,...,xn) ; argsI = (y1,...,ym)

```
let soc_step soc argsI argsO env = -- where "env" holds variables
    values
 let env1 = args2params argsI soc.ins env in
```

Instrumenting soc

```
soc foo(i1, ..., im) returns (o1,..., on);
let
  ... -- some equations
tel
soc bar(...) returns(...);
let
  ...;
  (x1,...,xn) = foo(y1,...,ym); -- how to evaluate this expression?
  ...;
tel
```

- node bar is compiled into a soc, and:
  - soc.ins = (i1, ..., im) ; soc.out = (o1,..., on)
  - argsO = (x1,...,xn) ; argsI = (y1,...,ym)

```
let soc_step soc argsI argsO env = -- where "env" holds variables
    values
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 in
```

Instrumenting soc

```
soc foo(i1, ..., im) returns (o1,..., on);
let
  ... -- some equations
tel
soc bar(...) returns(...);
let
  ...;
  (x1,...,xn) = foo(y1,...,ym); -- how to evaluate this expression?
  ...;
tel
```

- node bar is compiled into a soc, and:
  - soc.ins = (i1, ..., im) ; soc.out = (o1,..., on)
  - argsO = (x1,...,xn) ; argsI = (y1,...,ym)

```
let soc_step soc argsI argsO env = -- where "env" holds variables
    values
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 in
 let env3 = params2args soc.out argsO env2 in
```

Instrumenting soc

```
soc foo(i1, ..., im) returns (o1,..., on);
let
  ... -- some equations
tel
soc bar(...) returns(...);
let
  ...;
  (x1,...,xn) = foo(y1,...,ym); -- how to evaluate this expression?
  ...;
tel
```

- node bar is compiled into a soc, and:
  - soc.ins = (i1, ..., im) ; soc.out = (o1,..., on)
  - args0 = (x1,...,xn) ; argsI = (y1,...,ym)

```
let soc_step soc argsI args0 env = -- where "env" holds variables
    values
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 in
 let env3 = params2args soc.out args0 env2 in
 env3
```

A CPS soc interpreter to add events (1/2)

```
let soc_step soc argsI argsO env =
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 in
 let env3 = params2args soc.out argsO env2 in
 env3
```

A CPS soc interpreter to add events (1/2)

```
let soc_step soc argsI argsO env =
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 in
 let env3 = params2args soc.out argsO env2 in
 env3
```

```
let step soc argsI argsO env (cont:  env -> 'a):'a =
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 in
 let env3 = params2args soc.out argsO env2 in
 cont env3
```

A CPS soc interpreter to add events (1/2)

```
let soc_step soc argsI argsO env =
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 in
 let env3 = params2args soc.out argsO env2 in
 env3
```

```
let step soc argsI argsO env (cont:  env -> 'a):'a =
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 in
 let env3 = params2args soc.out argsO env2 in
 cont env3
```

```
let step soc argsI argsO env (cont:  env -> Event.t):Event.t =
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 in
 let env3 = params2args soc.out argsO env2 in
 {
    name = soc.name;
    kind = Exit; data = get_data env3;
    next = (fun () -> cont env3);
 }
```

A CPS soc interpreter to add events (2/2)

```
let step soc argsI argsO env cont =
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 (* goal: expose env1 HERE *) in
 let env3 = params2args soc.out argsO env2 in
 cont env3
```

A CPS soc interpreter to add events (2/2)

```
let step soc argsI argsO env cont =
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 (* goal: expose env1 HERE *) in
 let env3 = params2args soc.out argsO env2 in
 cont env3
```

```
let soc_step soc invals argsI argsO env cont =
  let env1 = arg2params argsI soc.ins env in
  let cont2 env2 = (* We introduce a fake function call HERE *)
    let env3 = param2args soc.out argsO env2 in
    cont env3
  in
  cont2 (do_the_step soc env1)
```

A CPS soc interpreter to add events (2/2)

```
let step soc argsI argsO env cont =
 let env1 = args2params argsI soc.ins env in
 let env2 = do_the_step soc env1 (* goal: expose env1 HERE *) in
 let env3 = params2args soc.out argsO env2 in
 cont env3
```

```
let soc_step soc invals argsI argsO env cont =
  let env1 = arg2params argsI soc.ins env in
  let cont2 env2 = (* We introduce a fake function call HERE *)
    let env3 = param2args soc.out argsO env2 in
    cont env3
  in
  cont2 (do_the_step soc env1)
```

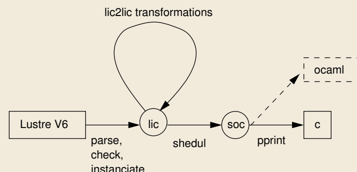
```
let soc_step soc invals argsI argsO env cont =
  let env1 = arg2params argsI soc.ins env in
  let cont2 env2 =
    let env3 = param2args soc.out argsO env2 in
    cont env3
  in
  {  name = soc.name; kind = Call; data = get_data env1;
     next = fun()-> cont2 (do_the_step soc env1) }
```

A CPS soc interpreter to add events (epilogue)

All together

```
let soc_step soc invals argsI argsO env cont =
  let env1 = arg2params argsI soc.ins env in
  let cont2 env2 =
    let env3 = param2args soc.out argsO env2 in
    {
      name = soc.name;
      kind = Exit; data = get_data env3;
      next = fun () -> cont env3;
    }
  in
  {
    name = soc.name;
    kind = Call; data = get_data env1;
    next = (fun () -> cont2 (do_the_step soc env1));
  }
```

Instrumenting generated code



- Would be more efficient
- Instrumenting C code
  - Require Interfacing C/Ocaml (easy)
  - Have C code in C (possible but less easy)
- Instrumenting Ocaml code
  - need a Ocaml Soc pretty-printer
  - still subject to divergence

Instrumenting ocaml code

The binding between args and params in done by the host language

```
let (n1_step: ti1 * ... *tim -> to1 * ... * ton) =
fun (i1, ..., im)
  let v1, o1 = n2_step(i1) in
  ...
  let on = n4_step(v3,im) in
  (o1, ..., on)
```

We'd need a step of type

```
step_dbg : (sl -> e -> ( sl -> e -> e) -> e);
```

Encapsulate the generate node

Encapsulate the generate node into a function with the rigth profile

```
let n1_step_dbg : (sl -> e -> ( sl -> e -> e) -> e) =
 fun sl e cont ->
  let (i1, ..., im) = get_n2_inputs sl in
  let (o1, ..., on) = n2_step(i1, ..., im) in
  let sl = set_n2_outputs (o1, ..., on) in
  cont sl e
```

Adding events

Adding events is then similar to the soc interpreter instrumentation.

```
let n2_step_dbg sl e cont  =
  let (i1, ..., im) = get_n2_inputs sl in
  let cont2 sl2 =
    let (o1, ..., on) = n2_step(i1, ..., im) in
    let sl = set_n2_outputs (o1, ..., on) in
      let sl3 = param2args soc.out args0 sl2 in
    {
      name = soc.name;
      kind = Exit;
      next = fun () -> cont sl3;
    }
  in
  {
    name = soc.name;
    kind = Call;
    data = sl;
    next = (fun () ->  cont2 (do_the_step soc sl1));
  }
```

Adding events

Adding events is then similar to the soc interpreter instrumentation.

```
let n2_step_dbg sl e cont  =
  let (i1, ..., im) = get_n2_inputs sl in
  let cont2 sl2 =
    let (o1, ..., on) = n2_step(i1, ..., im) in
    let sl = set_n2_outputs (o1, ..., on) in
      let sl3 = param2args soc.out args0 sl2 in
    {
      name = soc.name;
      kind = Exit;
      next = fun () -> cont sl3;
    }
  in
  {
    name = soc.name;
    kind = Call;
    data = sl;
    next = (fun () ->  cont2 (do_the_step soc sl1));
  }
```

nb: for inner node calls, we need a deeper instrumentation

Plan

1. rdbg : a programmable debugger for reactive programs
2. A motivating demo (Level 1)
3. Only a tiny API is necessary (Level 2)
4. Design choices for the kernel (Level 4)
5. Runtime Instrumentation (Level 3)
6. Conclusion

Performance

- Cost of the instrumentation
  - Less than 1% of penalty Lutin programs
  - 20 % of penalty on the Lustre soc interpreter

Performance

- Cost of the instrumentation
  - Less than 1% of penalty Lutin programs
  - 20 % of penalty on the Lustre soc interpreter
- bytecode versus native code => x3

Performance

- Cost of the instrumentation
  - Less than 1% of penalty Lutin programs
  - 20 % of penalty on the Lustre soc interpreter

- bytecode versus native code => x3

- The Lustre interpreter is 3 order of magnitude slower than the C code (hence the idea mentionned earlier of instrumenting some generated ocaml code)

Performance

- Cost of the instrumentation
  - Less than 1% of penalty Lutin programs
  - 20 % of penalty on the Lustre soc interpreter
- bytecode versus native code => x3
- The Lustre interpreter is 3 order of magnitude slower than the C code (hence the idea mentionned earlier of instrumenting some generated ocaml code)

But note that all this is generally dominated by the cost of the Lutin environment!

About `rdbg` design choices

- Cons
  - Requires an Ocaml interface
  - Requires CPS

About `rdbg` design choices

- Cons
  - Requires an Ocaml interface
  - Requires CPS
- Pros
  - Lightweight implementation (reuse ocaml libs and REPL)

About `rdbg` design choices

- Cons
  - Requires an Ocaml interface
  - Requires CPS
- Pros
  - Lightweight implementation (reuse ocaml libs and REPL)
  - Programmable!

About `rdbg` design choices

- Cons
  - Requires an Ocaml interface
  - Requires CPS
- Pros
  - Lightweight implementation (reuse ocaml libs and REPL)
  - Programmable!
  - Separation of concerns

About `rdbg` design choices

- Cons
  - Requires an Ocaml interface
  - Requires CPS
- Pros
  - Lightweight implementation (reuse ocaml libs and REPL)
  - Programmable!
  - Separation of concerns
    - a simple plugin API : `step_dbg`

About `rdbg` design choices

- Cons
  - Requires an Ocaml interface
  - Requires CPS
- Pros
  - Lightweight implementation (reuse ocaml libs and REPL)
  - Programmable!
  - Separation of concerns
    - a simple plugin API : `step_dbg`
    - a tiny kernel: `run : Rdbg.args -> Event.t`
    - a rich set of commands easily build on top of it
    - which could be reused on other languages
  - same tool for test and debug (debug oracles !)

The end

- Install

```
$ opam repo add verimag-sync-repo
    "http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/opam-repository"
$ opam install rdbg lustre-v6 lutin
```

The end

- Install

```
$ opam repo add verimag-sync-repo
    "http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/opam-repository"
$ opam install rdbg lustre-v6 lutin
```

Thanks for your attention